Eloquent JavaScript

THIRD EDITION

A Modern Introduction to Programming

Marijn Haverbeke

no starch press

Eloquent JavaScript

3ra edición

Marijn Haverbeke

Copyright © 2018 por Marijn Haverbeke

Este libro está bajo la licencia Creative Commons Atribución-NoComercial 3.0 (http://creativecommons.org/licenses/by-nc/3.0/). El código también puede considerarse bajo una licencia MIT (http://opensource.org/licenses/MIT).

Las ilustraciones han sido contribuidas por varios artistas: Cover and chapter illustrations by Madalina Tantareanu. Pixel art in Chapters 7 and 16 by Antonio Perdomo Pastor. Regular expression diagrams in Chapter 9 generated with regexper.com by Jeff Avallone. Village photograph in Chapter 11 by Fabrice Creuzot. Game concept for Chapter 15 by Thomas Palef.

The third edition of Eloquent JavaScript was made possible by 325 financial backers.

You can buy a print version of this book, with an extra bonus chapter included, printed by No Starch Press at http://a-fwd.com/com=marijhaver-20&asincom=1593279507.

Contents

Introducción 1
Acerca de la programación . . . 2
Por qué el lenguaje importa . . . 3
¿Qué es JavaScript? . . . 6
Código, y qué hacer con él . . . 8
Descripción general de este libro . . . 9
Convenciones tipográficas . . . 9

1 Valores, Tipos, y Operadores 11
Valores . . . 11
Números . . . 12
Strings . . . 15
Operadores unarios . . . 17
Valores Booleanos . . . 17
Valores vacíos . . . 20
Conversión de tipo automática . . . 20
Resumen . . . 22

2 Estructura de Programa 23
Expresiones y declaraciones . . . 23
Vinculaciones . . . 24
Nombres vinculantes . . . 26
El entorno . . . 27
Funciones . . . 27
La función console.log . . . 28
Valores de retorno . . . 28
Flujo de control . . . 29
Ejecución condicional . . . 29
Ciclos while y do . . . 31
Indentando Código . . . 33
Ciclos for . . . 34
Rompiendo un ciclo . . . 35

Actualizando vinculaciones de manera sucinta 35
Despachar en un valor con switch 36
Capitalización 37
Comentarios 38
Resumen 38
Ejercicios 39

3 Funciones 41
Definiendo una función 41
Vinculaciones y alcances 42
Funciones como valores 45
Notación de declaración 45
Funciones de flecha 46
La pila de llamadas 47
Argumentos Opcionales 48
Cierre 50
Recursión 51
Funciones crecientes 54
Funciones y efectos secundarios 57
Resumen 58
Ejercicios 58

4 Estructuras de Datos: Objetos y Arrays 60
El Hombre Ardilla 60
Conjuntos de datos 61
Propiedades 62
Métodos 63
Objetos 64
Mutabilidad 66
El diario del licántropo 68
Calculando correlación 70
Ciclos de array 71
El análisis final 72
Arrayología avanzada 74
Strings y sus propiedades 76
Parámetros restantes 77
El objeto Math 78
Desestructurar 80
JSON 81
Resumen 82

Ejercicios 83

5 Funciones de Orden Superior 86
Abstracción 87
Abstrayendo la repetición 88
Funciones de orden superior 89
Conjunto de datos de códigos 90
Filtrando arrays 91
Transformando con map 92
Resumiendo con reduce 93
Composabilidad 94
Strings y códigos de caracteres 96
Reconociendo texto 98
Resumen 99
Ejercicios 100

6 La Vida Secreta de los Objetos 101
Encapsulación 101
Métodos 102
Prototipos 103
Clases 105
Notación de clase 106
Sobreescribiendo propiedades derivadas 107
Mapas 109
Polimorfismo 111
Símbolos 111
La interfaz de iterador 113
Getters, setters y estáticos 115
Herencia 117
El operador instanceof 118
Resumen 119
Ejercicios 120

7 Proyecto: Un Robot 122
VillaPradera 122
La tarea 124
Datos persistentes 126
Simulación 127
La ruta del camión de correos 129
Búsqueda de rutas 129

Ejercicios 132

8 Bugs y Errores 134
Lenguaje 134
Modo estricto 135
Tipos 136
Probando 137
Depuración 138
Propagación de errores 140
Excepciones 141
Limpiando después de excepciones 143
Captura selectiva 145
Afirmaciones 147
Resumen 148
Ejercicios 148

9 Expresiones Regulares 150
Creando una expresión regular 150
Probando por coincidencias 151
Conjuntos de caracteres 151
Repitiendo partes de un patrón 153
Agrupando subexpresiones 154
Coincidencias y grupos 154
La clase Date (“Fecha”) 156
Palabra y límites de string 157
Patrones de elección 158
Las mecánicas del emparejamiento 158
Retrocediendo 159
El método replace 161
Codicia 163
Creando objetos RegExp dinámicamente 164
El método search 165
La propiedad lastIndex 166
Análisis de un archivo INI 168
Caracteres internacionales 170
Resumen 171
Ejercicios 173

10 Módulos 175
Módulos 175

Paquetes 176
Módulos improvisados 177
Evaluando datos como código 178
CommonJS 179
Módulos ECMAScript 182
Construyendo y empaquetando 183
Diseño de módulos 184
Resumen 186
Ejercicios 187

11 Programación Asincrónica 189
Asincronicidad 189
Tecnología cuervo 191
Devolución de llamadas 192
Promesas 194
Fracaso 196
Las redes son difíciles 197
Colecciones de promesas 200
Inundación de red 201
Enrutamiento de mensajes 202
Funciones asíncronas 205
Generadores 207
El ciclo de evento 208
Errores asincrónicos 209
Resumen 211
Ejercicios 211

12 Proyecto: Un Lenguaje de Programación 213
Análisis 213
The evaluator 218
Special forms 219
The environment 221
Functions 222
Compilation 223
Cheating 224
Exercises 225

13 El Modelo de Objeto del Documento 227
La estructura del documento 227
Árboles 228

El estándar . . . 229
Moviéndose a través del árbol . . . 230
Buscar elementos . . . 232
Actualizar el documento . . . 233
Crear nodos . . . 233
Atributos . . . 235
Layout . . . 236
Estilización . . . 238
Estilos en Cascada . . . 240
Selectores de consulta . . . 241
Posicionamiento y animaciones . . . 242
Resumen . . . 245
Ejercicios . . . 245

14 Manejo de Eventos 248
Manejador de eventos . . . 248
Eventos y nodos DOM . . . 249
Objetos de evento . . . 250
Propagación . . . 251
Acciones por defecto . . . 252
Eventos de teclado . . . 253
Eventos de puntero . . . 255
Eventos de desplazamiento . . . 259
Eventos de foco . . . 260
Evento de carga . . . 261
Eventos y el ciclo de eventos . . . 261
Temporizadores . . . 262
Antirrebote . . . 263
Resumen . . . 265
Ejercicios . . . 265

15 Proyecto: Un Juego de Plataforma 267
El juego . . . 267
La tecnología . . . 268
Niveles . . . 268
Leyendo un nivel . . . 269
Actors . . . 271
Encapsulation as a burden . . . 275
Drawing . . . 275
Motion and collision . . . 281

Actor updates . . . 284
Tracking keys . . . 286
Running the game . . . 286
Exercises . . . 289

16 Node.js 290
Antecedentes . . . 290
El comando Node . . . 291
Módulos . . . 292
Instalación con NPM . . . 293
El módulo del sistema de archivos . . . 296
The HTTP module . . . 297
Streams . . . 299
A file server . . . 301
Summary . . . 306
Exercises . . . 307

17 Proyecto: Sitio web para compartir habilidades 309
Diseño . . . 309
Long Polling . . . 310
HTTP interface . . . 311
The server . . . 314
The client . . . 321
Exercises . . . 327

Exercise Hints 329
Estructura de Programa . . . 329
Funciones . . . 330
Estructuras de Datos: Objetos y Arrays . . . 331
Funciones de Orden Superior . . . 333
La Vida Secreta de los Objetos . . . 334
Proyecto: Un Robot . . . 335
Bugs y Errores . . . 336
Expresiones Regulares . . . 336
Módulos . . . 337
Programación Asincrónica . . . 339
Proyecto: Un Lenguaje de Programación . . . 340
El Modelo de Objeto del Documento . . . 341
Manejo de Eventos . . . 342
Proyecto: Un Juego de Plataforma . . . 343

Node.js 344
Proyecto: Sitio web para compartir habilidades 345

"Nosotros creemos que estamos creando el sistema para nuestros propios propósitos. Creemos que lo estamos haciendo a nuestra propia imagen... Pero la computadora no es realmente como nosotros. Es una proyección de una parte muy delgada de nosotros mismos: esa porción dedicada a la lógica, el orden, la reglas y la claridad."

—Ellen Ullman, Close to the Machine: Technophilia and its Discontents

Introducción

Este es un libro acerca de instruir computadoras. Hoy en dia las computadoras son tan comunes como los destornilladores (aunque bastante más complejas que estos), y hacer que hagan exactamente lo que quieres que hagan no siempre es fácil.

Si la tarea que tienes para tu computadora es común, y bien entendida, tal y como mostrarte tu correo electrónico o funcionar como una calculadora, puedes abrir la aplicación apropiada y ponerte a trabajar en ella. Pero para realizar tareas únicas o abiertas, es posible que no haya una aplicación disponible.

Ahí es donde la programación podría entrar en juego. La *programación* es el acto de construir un *programa*—un conjunto de instrucciones precisas que le dicen a una computadora qué hacer. Porque las computadoras son bestias tontas y pedantes, la programación es fundamentalmente tediosa y frustrante.

Afortunadamente, si puedes superar eso, y tal vez incluso disfrutar el rigor de pensar en términos que las máquinas tontas puedan manejar, la programación puede ser muy gratificante. Te permite hacer en segundos cosas que tardarían *siglos* a mano. Es una forma de hacer que tu herramienta computadora haga cosas que antes no podía. Ademas proporciona de un maravilloso ejercicio en pensamiento abstracto.

La mayoría de la programación se realiza con lenguajes de programación. Un *lenguaje de programación* es un lenguaje artificialmente construido que se utiliza para instruir ordenadores. Es interesante que la forma más efectiva que hemos encontrado para comunicarnos con una computadora es bastante parecida a la forma que usamos para comunicarnos entre nosotros. Al igual que los lenguajes humanos, los lenguajes de computación permiten que las palabras y frases sean combinadas de nuevas maneras, lo que nos permite expresar siempre nuevos conceptos.

Las interfaces basadas en lenguajes, que en un momento fueron la principal forma de interactuar con las computadoras para la mayoría de las personas, han sido en gran parte reemplazadas con interfaces más simples y limitadas. Pero todavía están allí, si sabes dónde mirar.

En un punto, las interfaces basadas en lenguajes, como las terminales BASIC

y DOS de los 80 y 90, eran la principal forma de interactuar con las computadoras. Estas han sido reemplazados en gran medida por interfaces visuales, las cuales son más fáciles de aprender pero ofrecen menos libertad. Los lenguajes de computadora todavía están allí, si sabes dónde mirar. Uno de esos lenguajes, JavaScript, está integrado en cada navegador web moderno y, por lo tanto, está disponible en casi todos los dispositivos.

Este libro intentará familiarizarte lo suficiente con este lenguaje para poder hacer cosas útiles y divertidas con él.

Acerca de la programación

Además de explicar JavaScript, también introduciré los principios básicos de la programación. La programación, en realidad, es difícil. Las reglas fundamentales son típicamente simples y claras, pero los programas construidos en base a estas reglas tienden a ser lo suficientemente complejas como para introducir sus propias reglas y complejidad. De alguna manera, estás construyendo tu propio laberinto, y es posible que te pierdas en él.

Habrá momentos en los que leer este libro se sentirá terriblemente frustrante. Si eres nuevo en la programación, habrá mucho material nuevo para digerir. Gran parte de este material sera entonces *combinado* en formas que requerirán que hagas conexiones adicionales.

Depende de ti hacer el esfuerzo necesario. Cuando estés luchando para seguir el libro, no saltes a ninguna conclusión acerca de tus propias capacidades. Estás bien, sólo tienes que seguir intentando. Tomate un descanso, vuelve a leer algún material, y asegúrate de leer y comprender los programas de ejemplo y ejercicios. Aprender es un trabajo duro, pero todo lo que aprendes se convertirá en tuyo, y hará que el aprendizaje subsiguiente sea más fácil.

> Cuando la acción deja de servirte, reúne información; cuando la información deja de servirte, duerme..
>
> —Ursula K. Le Guin, La Mano Izquierda De La Oscuridad

Un programa son muchas cosas. Es una pieza de texto escrita por un programador, es la fuerza directriz que hace que la computadora haga lo que hace, son datos en la memoria de la computadora, y sin embargo controla las acciones realizadas en esta misma memoria. Las analogías que intentan comparar programas a objetos con los que estamos familiarizados tienden a fallar. Una analogía que es superficialmente adecuada es el de una máquina —muchas partes separadas tienden a estar involucradas—, y para hacer que todo funcione, tenemos

que considerar la formas en las que estas partes se interconectan y contribuyen a la operación de un todo.

Una computadora es una máquina física que actúa como un anfitrión para estas máquinas inmateriales. Las computadoras en si mismas solo pueden hacer cosas estúpidamente sencillas. La razón por la que son tan útiles es que hacen estas cosas a una velocidad increíblemente alta. Un programa puede ingeniosamente combinar una cantidad enorme de estas acciones simples para realizar cosas bastante complicadas.

Un programa es un edificio de pensamiento. No cuesta nada construirlo, no pesa nada, y crece fácilmente bajo el teclear de nuestras manos.

Pero sin ningún cuidado, el tamaño de un programa y su complejidad crecerán sin control, confundiendo incluso a la persona que lo creó. Mantener programas bajo control es el problema principal de la programación. Cuando un programa funciona, es hermoso. El arte de la programación es la habilidad de controlar la complejidad. Un gran programa es moderado, hecho simple en su complejidad.

Algunos programadores creen que esta complejidad se maneja mejor mediante el uso de solo un pequeño conjunto de técnicas bien entendidas en sus programas. Ellos han compuesto reglas estrictas ("mejores prácticas") que prescriben la forma que los programas deberían tener, y se mantienen cuidadosamente dentro de su pequeña y segura zona.

Esto no solamente es aburrido, sino que también es ineficaz. Problemas nuevos a menudo requieren soluciones nuevas. El campo de la programación es joven y todavía se esta desarrollando rápidamente, y es lo suficientemente variado como para tener espacio para aproximaciones salvajemente diferentes. Hay muchos errores terribles que hacer en el diseño de programas, así que ve adelante y comételos para que los entiendas mejor. La idea de cómo se ve un buen programa se desarrolla con la práctica, no se aprende de una lista de reglas.

Por qué el lenguaje importa

Al principio, en el nacimiento de la informática, no habían lenguajes de programación. Los programas se veían mas o menos así:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
```

```
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Ese es un programa que suma los números del 1 al 10 entre ellos e imprime el resultado: `1 + 2 + ... + 10 = 55`. Podría ser ejecutado en una simple máquina hipotética. Para programar las primeras computadoras, era necesario colocar grandes arreglos de interruptores en la posición correcta o perforar agujeros en tarjetas de cartón y dárselos a la computadora. Probablemente puedas imaginarte lo tedioso y propenso a errores que era este procedimiento. Incluso escribir programas simples requería de mucha inteligencia y disciplina. Los complejos eran casi inconcebibles.

Por supuesto, ingresar manualmente estos patrones arcanos de bits (los unos y ceros) le dieron al programador un profundo sentido de ser un poderoso mago. Y eso tiene que valer algo en términos de satisfacción laboral.

{{index memoria, instrucción}}

Cada línea del programa anterior contiene una sola instrucción. Podría ser escrito en español así:

1. Almacenar el número 0 en la ubicación de memoria 0.
2. Almacenar el número 1 en la ubicación de memoria 1.
3. Almacenar el valor de la ubicación de memoria 1 en la ubicación de memoria 2.
4. Restar el número 11 del valor en la ubicación de memoria 2.
5. Si el valor en la ubicación de memoria 2 es el número 0, continuar con la instrucción 9.
6. Sumar el valor de la ubicación de memoria 1 a la ubicación de memoria 0.
7. Sumar el número 1 al valor de la ubicación de memoria 1.
8. Continuar con la instrucción 3.
9. Imprimir el valor de la ubicación de memoria 0.

Aunque eso ya es más legible que la sopa de bits, es aún difícil de entender. Usar nombres en lugar de números para las instrucciones y ubicaciones de memoria ayuda:

```
 Establecer "total" como 0.
 Establecer "cuenta" como 1.
[loop]
 Establecer "comparar" como "cuenta".
 Restar 11 de "comparar".
 Si "comparar" es cero, continuar en [fin].
 Agregar "cuenta" a "total".
 Agregar 1 a "cuenta".
 Continuar en [loop].
[fin]
 Imprimir "total".
```

¿Puedes ver cómo funciona el programa en este punto? Las primeras dos líneas le dan a dos ubicaciones de memoria sus valores iniciales: se usará `total` para construir el resultado de la computación, y `cuenta` hará un seguimiento del número que estamos mirando actualmente. Las líneas usando `comparar` son probablemente las más extrañas. El programa quiere ver si `cuenta` es igual a 11 para decidir si puede detener su ejecución. Debido a que nuestra máquina hipotética es bastante primitiva, esta solo puede probar si un número es cero y hace una decisión (o salta) basándose en eso. Por lo tanto, usa la ubicación de memoria etiquetada como `comparar` para calcular el valor de `cuenta - 11` y toma una decisión basada en ese valor. Las siguientes dos líneas agregan el valor de `cuenta` al resultado e incrementan `cuenta` en 1 cada vez que el programa haya decidido que `cuenta` todavía no es 11.

Aquí está el mismo programa en JavaScript:

```
let total = 0, cuenta = 1;
while (cuenta <= 10) {
  total += cuenta;
  cuenta += 1;
}
console.log(total);
// → 55
```

Esta versión nos da algunas mejoras más. La más importante, ya no hay necesidad de especificar la forma en que queremos que el programa salte hacia adelante y hacia atrás. El constructo del lenguaje `while` se ocupa de eso. Este continúa ejecutando el bloque de código (envuelto en llaves) debajo de el, siempre y cuando la condición que se le dio se mantenga. Esa condición es `cuenta <= 10`, lo que significa "*cuenta* es menor o igual a 10". Ya no tenemos que crear un valor temporal y compararlo con cero, lo cual era un detalle

poco interesante. Parte del poder de los lenguajes de programación es que se encargan por nosotros de los detalles sin interés.

Al final del programa, después de que el `while` haya terminado, la operación `console.log` se usa para mostrar el resultado.

{{index “sum function”, “range function”, abstracción, function}}

Finalmente, aquí está cómo se vería el programa si tuviéramos acceso a las las convenientes operaciones `rango` y `suma` disponibles, que respectivamente crean una colección de números dentro de un rango y calculan la suma de una colección de números:

```
console.log(suma(rango(1, 10)));
// → 55
```

La moraleja de esta historia es que el mismo programa se puede expresar en formas largas y cortas, ilegibles y legibles. La primera versión del programa era extremadamente oscura, mientras que esta última es casi Español: muestra en el `log` de la consola la `suma` del `rango` de los números 1 al 10. (En capítulos posteriores veremos cómo definir operaciones como `suma` y `rango`.)

Un buen lenguaje de programación ayuda al programador permitiéndole hablar sobre las acciones que la computadora tiene que realizar en un nivel superior. Ayuda a omitir detalles poco interesantes, proporciona bloques de construcción convenientes (como `while` y `console.log`), te permite que definas tus propios bloques de construcción (como `suma` y `rango`), y hace que esos bloques sean fáciles de componer.

¿Qué es JavaScript?

JavaScript se introdujo en 1995 como una forma de agregar programas a páginas web en el navegador Netscape Navigator. El lenguaje ha sido desde entonces adoptado por todos los otros navegadores web principales. Ha hecho que las aplicaciones web modernas sean posibles: aplicaciones con las que puedes interactuar directamente, sin hacer una recarga de página para cada acción. JavaScript también es utilizado en sitios web más tradicionales para proporcionar diversas formas de interactividad e ingenio.

Es importante tener en cuenta que JavaScript casi no tiene nada que ver con el lenguaje de programación llamado Java. El nombre similar fue inspirado por consideraciones de marketing, en lugar de buen juicio. Cuando JavaScript estaba siendo introducido, el lenguaje Java estaba siendo fuertemente comercializado y estaba ganando popularidad. Alguien pensó que era una buena idea

intentar cabalgar sobre este éxito. Ahora estamos atrapados con el nombre.

Después de su adopción fuera de Netscape, un documento estándar fue escrito para describir la forma en que debería funcionar el lenguaje JavaScript, para que las diversas piezas de software que decían ser compatibles con JavaScript en realidad estuvieran hablando del mismo lenguaje. Este se llamo el Estándar ECMAScript, por Ecma International que hizo la estandarización. En la práctica, los términos ECMAScript y JavaScript se puede usar indistintamente, son dos nombres para el mismo lenguaje.

Hay quienes dirán cosas *terribles* sobre JavaScript. Muchas de estas cosas son verdaderas. Cuando estaba comenzando a escribir algo en JavaScript por primera vez, rápidamente comencé a despreciarlo. El lenguaje aceptaba casi cualquier cosa que escribiera, pero la interpretaba de una manera que era completamente diferente de lo que quería decir. Por supuesto, esto tenía mucho que ver con el hecho de que no tenía idea de lo que estaba haciendo, pero hay un problema real aquí: JavaScript es ridículamente liberal en lo que permite. La idea detrás de este diseño era que haría a la programación en JavaScript más fácil para los principiantes. En realidad, lo que mas hace es que encontrar problemas en tus programas sea más difícil porque el sistema no los señalará por ti.

Sin embargo, esta flexibilidad también tiene sus ventajas. Deja espacio para muchas técnicas que son imposibles en idiomas más rígidos, y como verás (por ejemplo en el Capítulo 10) se pueden usar para superar algunas de las deficiencias de JavaScript. Después de aprender el idioma correctamente y luego de trabajar con él por un tiempo, he aprendido a *querer* a JavaScript.

Ha habido varias versiones de JavaScript. ECMAScript versión 3 fue la versión mas ampliamente compatible en el momento del ascenso de JavaScript a su dominio, aproximadamente entre 2000 y 2010. Durante este tiempo, se trabajó en marcha hacia una ambiciosa versión 4, que planeaba una serie de radicales mejoras y extensiones al lenguaje. Cambiar un lenguaje vivo y ampliamente utilizado de una manera tan radical resultó ser políticamente difícil, y el trabajo en la versión 4 fue abandonado en 2008, lo que llevó a la versión 5, mucho menos ambiciosa, que se publicaría en el 2009. Luego, en 2015, una actualización importante, incluyendo algunas de las ideas planificadas para la versión 4, fue realizada. Desde entonces hemos tenido actualizaciones nuevas y pequeñas cada año.

El hecho de que el lenguaje esté evolucionando significa que los navegadores deben mantenerse constantemente al día, y si estás usando uno más antiguo, puede que este no soporte todas las mejoras. Los diseñadores de lenguajes tienen cuidado de no realizar cualquier cambio que pueda romper los programas ya existentes, de manera que los nuevos navegadores puedan todavía ejecutar

programas viejos. En este libro, usaré la versión 2017 de JavaScript.

Los navegadores web no son las únicas plataformas en las que se usa JavaScript. Algunas bases de datos, como MongoDB y CouchDB, usan JavaScript como su lenguaje de scripting y consultas. Varias plataformas para programación de escritorio y servidores, más notablemente el proyecto Node.js (el tema del Capítulo 20) proporcionan un entorno para programar en JavaScript fuera del navegador.

Código, y qué hacer con él

Código es el texto que compone los programas. La mayoría de los capítulos en este libro contienen bastante. Creo que leer código y escribir código son partes indispensables del aprendizaje para programar. Trata de no solo echar un vistazo a los ejemplos, léelos atentamente y entiéndelos. Esto puede ser algo lento y confuso al principio, pero te prometo que rápidamente vas agarrar el truco. Lo mismo ocurre con los ejercicios. No supongas que los entiendes hasta que hayas escrito una solución funcional para resolverlos.

Te recomiendo que pruebes tus soluciones a los ejercicios en un intérprete real de JavaScript. De esta forma, obtendrás retroalimentación inmediata acerca de que si esta funcionando lo que estás haciendo, y, espero, serás tentado a experimentar e ir más allá de los ejercicios.

La forma más fácil de ejecutar el código de ejemplo en el libro y experimentar con él, es buscarlo en la versión en línea del libro en *eloquentjavascript.net*. Alli puedes hacer clic en cualquier ejemplo de código para editar y ejecutarlo y ver el resultado que produce. Para trabajar en los ejercicios, ve a *eloquentjavascript.net/code*, que proporciona el código de inicio para cada ejercicio de programación y te permite ver las soluciones.

Si deseas ejecutar los programas definidos en este libro fuera de la caja de arena del libro, se requiere cierto cuidado. Muchos ejemplos se mantienen por si mismos y deberían de funcionar en cualquier entorno de JavaScript. Pero código en capítulos más avanzados a menudo se escribe para un entorno específico (el navegador o Node.js) y solo puede ser ejecutado allí. Además, muchos capítulos definen programas más grandes, y las piezas de código que aparecen en ellos dependen de otras piezas o de archivos externos. La caja de arena en el sitio web proporciona enlaces a archivos Zip que contienen todos los scripts y archivos de datos necesarios para ejecutar el código de un capítulo determinado.

Descripción general de este libro

Este libro contiene aproximadamente tres partes. Los primeros 12 capítulos discuten el lenguaje JavaScript en sí. Los siguientes siete capítulos son acerca de los navegadores web y la forma en la que JavaScript es usado para programarlos. Finalmente, dos capítulos están dedicados a Node.js, otro entorno en donde programar JavaScript.

A lo largo del libro, hay cinco *capítulos de proyectos*, que describen programas de ejemplo más grandes para darte una idea de la programación real. En orden de aparición, trabajaremos en la construcción de un robot de delivery, un lenguaje de programación, un juego de plataforma, un programa de paint y un sitio web dinámico.

La parte del lenguaje del libro comienza con cuatro capítulos para presentar la estructura básica del lenguaje de JavaScript. Estos introducen estructuras de control (como la palabra `while` que ya viste en esta introducción), funciones (escribir tus propios bloques de construcción), y estructuras de datos. Después de estos, seras capaz de escribir programas simples. Luego, los Capítulos 5 y 6 introducen técnicas para usar funciones y objetos y asi escribir código más *abstracto* y de manera que puedas mantener la complejidad bajo control.

Después de un primer capítulo de proyecto, la primera parte del libro continúa con los capítulos sobre manejo y solución de errores, en expresiones regulares (una herramienta importante para trabajar con texto), en modularidad (otra defensa contra la complejidad), y en programación asincrónica (que se encarga de eventos que toman tiempo). El segundo capítulo de proyecto concluye la primera parte del libro.

La segunda parte, Capítulos 13 a 19, describe las herramientas a las que el JavaScript en un navegador tiene acceso. Aprenderás a mostrar cosas en la pantalla (Capítulos 14 y 17), responder a entradas de usuario (Capitulo 15), y a comunicarte a través de la red (Capitulo 18). Hay dos capítulos de proyectos en este parte.

Después de eso, el Capítulo 20 describe Node.js, y el Capitulo 21 construye un pequeño sistema web usando esta herramienta.

Convenciones tipográficas

En este libro, el texto escrito en una fuente `monoespaciada` representará elementos de programas, a veces son fragmentos autosuficientes, y a veces solo se refieren a partes de un programa cercano. Los programas (de los que ya has visto algunos), se escriben de la siguiente manera:

```
function factorial(numero) {
  if (numero == 0) {
    return 1;
  } else {
    return factorial(numero - 1) * numero;
  }
}
```

Algunas veces, para mostrar el resultado que produce un programa, la salida esperada se escribe después de el, con dos diagonales y una flecha en frente.

```
console.log(factorial(8));
// → 40320
```

¡Buena suerte!

> *"Debajo de la superficie de la máquina, el programa se mueve. Sin esfuerzo, se expande y se contrae. En gran armonía, los electrones se dispersan y se reagrupan. Las figuras en el monitor son tan solo ondas sobre el agua. La esencia se mantiene invisible debajo de la superficie."*
>
> —Master Yuan-Ma, The Book of Programming

Chapter 1

Valores, Tipos, y Operadores

Dentro del mundo de la computadora, solo existen datos. Puedes leer datos, modificar datos, crear nuevos datos—pero todo lo que no sean datos, no puede ser mencionado. Toda estos datos están almacenados como largas secuencias de bits, y por lo tanto, todos los datos son fundamentalmente parecidos.

Los *bits* son cualquier tipo de cosa que pueda tener dos valores, usualmente descritos como ceros y unos. Dentro de la computadora, estos toman formas tales como cargas eléctricas altas o bajas, una señal fuerte o débil, o un punto brillante u opaco en la superficie de un CD. Cualquier pedazo de información discreta puede ser reducida a una secuencia de ceros y unos y, de esa manera ser representada en bits.

Por ejemplo, podemos expresar el numero 13 en bits. Funciona de la misma manera que un número decimal, pero en vez de 10 diferentes dígitos, solo tienes 2, y el peso de cada uno aumenta por un factor de 2 de derecha a izquierda. Aquí tenemos los bits que conforman el número 13, con el peso de cada dígito mostrado debajo de el:

```
  0   0   0   0   1   1   0   1
128  64  32  16   8   4   2   1
```

Entonces ese es el número binario 00001101, o 8 + 4 + 1, o 13.

Valores

Imagina un mar de bits—un océano de ellos. Una computadora moderna promedio tiene mas de 30 billones de bits en su almacenamiento de datos volátiles (memoria funcional). El almacenamiento no volátil (disco duro o equivalente) tiende a tener unas cuantas mas ordenes de magnitud.

Para poder trabajar con tales cantidades de bits sin perdernos, debemos separarlos en porciones que representen pedazos de información. En un entorno de JavaScript, esas porciones son llamadas *valores*. Aunque todos los valores

están hechos de bits, estos juegan papeles diferentes. Cada valor tiene un tipo que determina su rol. Algunos valores son números, otros son pedazos de texto, otros son funciones, y asi sucesivamente.

Para crear un valor, solo debemos de invocar su nombre. Esto es conveniente. No tenemos que recopilar materiales de construcción para nuestros valores, o pagar por ellos. Solo llamamos su nombre, y *woosh*, ahi lo tienes. Estos no son realmente creados de la nada, por supuesto. Cada valor tiene que ser almacenado en algún sitio, y si quieres usar una cantidad gigante de valores al mismo tiempo, puede que te quedes sin memoria. Afortunadamente, esto solo es un problema si los necesitas todos al mismo tiempo. Tan pronto como dejes de utilizar un valor, este se disipará, dejando atrás sus bits para que estos sean reciclados como material de construcción para la próxima generación de valores.

Este capitulo introduce los elementos atómicos de los programas en JavaScript, estos son, los tipos de valores simples y los operadores que actúan en tales valores.

Números

Valores del tipo *number* (número) son, como es de esperar, valores numéricos. En un programa hecho en JavaScript, se escriben de la siguiente manera:

```
13
```

Utiliza eso en un programa, y ocasionara que el patron de bits que representa el número 13 sea creado dentro de la memoria de la computadora.

JavaScript utiliza un número fijo de bits, específicamente 64 de ellos, para almacenar un solo valor numérico. Solo existen una cantidad finita de patrones que podemos crear con 64 bits, lo que significa que la cantidad de números diferentes que pueden ser representados es limitada. Para una cantidad de N dígitos decimales, la cantidad de números que pueden ser representados es 10^{N}. Del mismo modo, dados 64 dígitos binarios, podemos representar 2^{64} números diferentes, lo que es alrededor de 18 mil trillones (un 18 con 18 ceros más). Eso es muchísimo.

La memoria de un computador solía ser mucho mas pequeña que en la actualidad, y las personas tendían a utilizar grupos de 8 o 16 bits para representar sus números. Era común accidentalmente *desbordar* esta limitación— terminando con un número que no cupiera dentro de la cantidad dada de bits. Hoy en día, incluso computadoras que caben dentro de tu bolsillo poseen de bastante

memoria, por lo que somos libres de usar pedazos de memoria de 64 bits, y solamente nos tenemos que preocupar por desbordamientos de memoria cuando lidiamos con números verdaderamente astronómicos.

A pesar de esto, no todos los números enteros por debajo de 18 mil trillones caben en un número de JavaScript. Esos bits también almacenan números negativos, por lo que un bit indica el signo de un número. Un problema mayor es que los números no enteros tienen que ser representados también. Para hacer esto, algunos de los bits son usados para almacenar la posición del punto decimal. El número entero mas grande que puede ser almacenado está en el rango de los 9 trillones (15 ceros)—lo cual es todavía placenteramente inmenso.

Los números fraccionarios se escriben usando un punto:

```
9.81
```

Para números muy grandes o muy pequeños, pudiéramos también usar notación científica agregando una *e* (de "exponente"), seguida por el exponente del número:

```
2.998e8
```

Eso es $2.998 \times 10^8 = 299{,}800{,}000$.

Los cálculos con números enteros (también llamados *integers*) mas pequeños a los 9 trillones anteriormente mencionados están garantizados a ser siempre precisos. Desafortunadamente, los calculos con números fraccionarios, generalmente no lo son. Así como π (pi) no puede ser precisamente expresado por un número finito de números decimales, muchos números pierden algo de precisión cuando solo hay 64 bits disponibles para almacenarlos. Esto es una pena, pero solo causa problemas prácticos en situaciones especificas. Lo importante es que debemos ser consciente de estas limitaciones y tratar a los números fraccionarios como aproximaciones, no como valores precisos.

Aritmética

Lo que mayormente se hace con los números es aritmética. Operaciones aritméticas tales como la adición y la multiplicación, toman dos valores numéricos y producen un nuevo valor a raíz de ellos. Asi es como lucen en JavaScript:

```
100 + 4 * 11
```

Los símbolos `+` y `*` son llamados *operadores*. El primero representa a la adición, y el segundo representa a la multiplicación. Colocar un operador entre dos valores aplicará la operación asociada a esos valores y producirá un nuevo valor.

¿Pero el ejemplo significa "agrega 4 y 100, y multiplica el resultado por 11", o es la multiplicación aplicada antes de la adición? Como quizás hayas podido adivinar, la multiplicación sucede primero. Pero asi como en las matemáticas, puedes cambiar este orden envolviendo la adición en paréntesis:

```
(100 + 4) * 11
```

Para sustraer, existe el operador `-`, y la división puede ser realizada con el operador `/`.

Cuando operadores aparecen juntos sin paréntesis, el orden en el cual son aplicados es determinado por la *precedencia* de los operadores. El ejemplo muestra que la multiplicación es aplicada antes que la adición. El operador `/` tiene la misma precedencia que `*`. Lo mismo aplica para `+` y `-`. Cuando operadores con la misma precedencia aparecen uno al lado del otro, como en `1 - 2 + 1`, estos se aplican de izquierda a derecha: `(1 - 2)+ 1`.

Estas reglas de precedencia no son algo de lo que deberias preocuparte. Cuando tengas dudas, solo agrega un paréntesis.

Existe otro operador aritmético que quizás no reconozcas inmediatamente. El símbolo `%` es utilizado para representar la operación de *residuo*. `X % Y` es el residuo de dividir `X` entre `Y`. Por ejemplo, `314 % 100` produce `14`, y `144 % 12` produce `0`. La precedencia del residuo es la la misma que la de la multiplicación y la división. Frecuentemente veras que este operador es tambien conocido como *modulo*.

Números especiales

Existen 3 valores especiales en JavaScript que son considerados números pero que no se comportan como números normales.

Los primeros dos son `Infinity` y `-Infinity`, los cuales representan las infinidades positivas y negativas. `Infinity - 1` aun es `Infinity`, y asi sucesivamente. A pesar de esto, no confíes mucho en computaciones que dependan de infinidades. Estas no son matemáticamente confiables, y puede que muy rápidamente nos resulten en el próximo número especial: `NaN`.

`NaN` significa "no es un número" ("Not A Number"), aunque *sea* un valor del tipo numérico. Obtendras este resultado cuando, por ejemplo, trates de

calcular `0 / 0` (cero dividido entre cero), `Infinity - Infinity`, o cualquier otra cantidad de operaciones numéricas que no produzcan un resultado significante.

Strings

El próximo tipo de dato básico es el *string*. Los Strings son usados para representar texto. Son escritos encerrando su contenido en comillas:

```
`Debajo en el mar`
"Descansa en el océano"
'Flota en el océano'
```

Puedes usar comillas simples, comillas dobles, o comillas invertidas para representar strings, siempre y cuando las comillas al principio y al final coincidan.

Casi todo puede ser colocado entre comillas, y JavaScript construirá un valor string a partir de ello. Pero algunos caracteres son mas difíciles. Te puedes imaginar que colocar comillas entre comillas podría ser difícil. Los *Newlines* (los caracteres que obtienes cuando presionas la tecla de Enter) solo pueden ser incluidos cuando el string está encapsulado con comillas invertidas (\`).

Para hacer posible incluir tales caracteres en un string, la siguiente notación es utilizada: cuando una barra invertida (\) es encontrada dentro de un texto entre comillas, indica que el carácter que le sigue tiene un significado especial. Esto se conoce como *escapar* el carácter. Una comilla que es precedida por una barra invertida no representará el final del string sino que formara parte del mismo. Cuando el carácter `n` es precedido por una barra invertida, este se interpreta como un Newline (salto de linea). De la mima forma, `t` después de una barra invertida, se interpreta como un character de tabulación. Toma como referencia el siguiente string:

```
"Esta es la primera linea\nY esta es la segunda"
```

El texto actual es este:

```
Esta es la primera linea
Y esta es la segunda
```

Se encuentran, por supuesto, situaciones donde queremos que una barra invertida en un string solo sea una barra invertida, y no un carácter especial. Si dos barras invertidas prosiguen una a la otra, serán colapsadas y sólo una per-

manecerá en el valor resultante del string. Asi es como el string "*Un carácter de salto de linea es escrito así: "\n".*" puede ser expresado:

```
Un carácter de salto de linea es escrito así: \"\\n\"."
```

También los strings deben de ser modelados como una serie de bits para poder existir dentro del computador. La forma en la que JavaScript hace esto es basada en el estándar *Unicode*. Este estándar asigna un número a todo carácter que alguna vez pudieras necesitar, incluyendo caracteres en Griego, Árabe, Japones, Armenio, y asi sucesivamente. Si tenemos un número para representar cada carácter, un string puede ser descrito como una secuencia de números.

Y eso es lo que hace JavaScript. Pero hay una complicación: La representación de JavaScript usa 16 bits por cada elemento string, en el cual caben 2^{16} números diferentes. Pero Unicode define mas caracteres que aquellos—aproximadamente el doble, en este momento. Entonces algunos caracteres, como muchos emojis, necesitan ocupar dos "posiciones de caracteres" en los strings de JavaScript. Volveremos a este tema en el Capitulo 5.

Los strings no pueden ser divididos, multiplicados, o substraidos, pero el operador `+` *puede* ser utilizado en ellos. No los agrega, sino que los *concatena*—pega dos strings juntos. La siguiente línea producirá el string `"concatenar"`:

```
"con" + "cat" + "e" + "nar"
```

Los valores string tienen un conjunto de funciones (*métodos*) asociadas, que pueden ser usadas para realizar operaciones en ellos. Regresaremos a estas en el Capítulo 4.

Los strings escritos con comillas simples o dobles se comportan casi de la misma manera—La unica diferencia es el tipo de comilla que necesitamos para escapar dentro de ellos. Los strings de comillas inversas, usualmente llamados *plantillas literales*, pueden realizar algunos trucos más. Mas alla de permitir saltos de lineas, pueden también incrustar otros valores.

```
`la mitad de 100 es ${100 / 2}`
```

Cuando escribes algo dentro de `${}` en una plantilla literal, el resultado será computado, convertido a string, e incluido en esa posición. El ejemplo anterior produce "*la mitad de 100 es 50*".

Operadores unarios

No todo los operadores son simbolos. Algunos se escriben como palabras. Un ejemplo es el operador `typeof`, que produce un string con el nombre del tipo de valor que le demos.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

Usaremos `console.log` en los ejemplos de código para indicar que que queremos ver el resultado de alguna evaluación. Mas acerca de esto esto en el proximo capitulo.

En los otros operadores que hemos visto hasta ahora, todos operaban en dos valores, pero `typeof` sola opera con un valor. Los operadores que usan dos valores son llamados operadores *binarios*, mientras que aquellos operadores que usan uno son llamados operadores *unarios*. El operador menos puede ser usado tanto como un operador binario o como un operador unario.

```
console.log(- (10 - 2))
// → -8
```

Valores Booleanos

Es frecuentemente util tener un valor que distingue entre solo dos posibilidades, como “si”, y “no”, o “encendido” y “apagado”. Para este propósito, JavaScript tiene el tipo *Boolean*, que tiene solo dos valores: true (verdadero) y false (falso) que se escriben de la misma forma.

Comparación

Aquí se muestra una forma de producir valores Booleanos:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

Los signos > y < son tradicionalmente símbolos para "mayor que" y "menor que", respectivamente. Ambos son operadores binarios. Aplicarlos resulta en un valor Boolean que indica si la condición que indican se cumple.

Los Strings pueden ser comparados de la misma forma.

```
console.log("Aardvark" < "Zoroaster")
// → true
```

La forma en la que los strings son ordenados, es aproximadamente alfabético, aunque no realmente de la misma forma que esperaríamos ver en un diccionario: las letras mayúsculas son siempre "menores que" las letras minúsculas, así que `"Z" < "a"`, y caracteres no alfabéticos (como `!`, `-` y demás) son también incluidos en el ordenamiento. Cuando comparamos strings, JavaScript evalúa los caracteres de izquierda a derecha, comparando los códigos Unicode uno por uno.

Otros operadores similares son `>=` (mayor o igual que), `<=` (menor o igual que), `==` (igual a), y `!=` (no igual a).

```
console.log("Itchy" != "Scratchy")
// → true
console.log("Manzana" == "Naranja")
// → false
```

Solo hay un valor en JavaScript que no es igual a si mismo, y este es `NaN` ("no es un número").

```
console.log(NaN == NaN)
// → false
```

Se supone que `NaN` denota el resultado de una computación sin sentido, y como tal, no es igual al resultado de ninguna *otra* computación sin sentido.

Operadores lógicos

También existen algunas operaciones que pueden ser aplicadas a valores Booleanos. JavaScript soporta tres operadores lógicos: *and*, *or*, y *not*. Estos pueden ser usados para "razonar" acerca de valores Booleanos.

El operador `&&` representa el operador lógico *and*. Es un operador binario, y su resultado es verdadero solo si ambos de los valores dados son verdaderos.

```
console.log(true && false)
```

```
// → false
console.log(true && true)
// → true
```

El operador `||` representa el operador lógico *or*. Lo que produce es verdadero si cualquiera de los valores dados es verdadero.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

Not se escribe como un signo de exclamación (`!`). Es un operador unario que voltea el valor dado—`!true` produce `false` y `!false` produce `true`.

Cuando estos operadores Booleanos son mezclados con aritmética y con otros operadores, no siempre es obvio cuando son necesarios los paréntesis. En la práctica, usualmente puedes manejarte bien sabiendo que de los operadores que hemos visto hasta ahora, `||` tiene la menor precedencia, luego le sigue `&&`, luego le siguen los operadores de comparación (`>`, `==`, y demás), y luego el resto. Este orden ha sido determinado para que en expresiones como la siguiente, la menor cantidad de paréntesis posible sea necesaria:

```
1 + 1 == 2 && 10 * 10 > 50
```

El ultimo operador lógico que discutiremos no es unario, tampoco binario, sino *ternario*, esto es, que opera en tres valores. Es escrito con un signo de interrogación y dos puntos, de esta forma:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

Este es llamado el operador *condicional* (o algunas veces simplemente operador *ternario* ya que solo existe uno de este tipo). El valor a la izquierda del signo de interrogación "decide" cual de los otros dos valores sera retornado. Cuando es verdadero, elige el valor de en medio, y cuando es falso, el valor de la derecha.

Valores vacíos

Existen dos valores especiales, escritos como `null` y `undefined`, que son usados para denotar la ausencia de un valor *significativo*. Son en si mismos valores, pero no traen consigo información.

Muchas operaciones en el lenguaje que no producen un valor significativo (veremos algunas mas adelante), producen `undefined` simplemente porque tienen que producir *algún* valor.

La diferencia en significado entre `undefined` y `null` es un accidente del diseño de JavaScript, y realmente no importa la mayor parte del tiempo. En los casos donde realmente tendríamos que preocuparnos por estos valores, mayormente recomiendo que los trates como intercambiables.

Conversión de tipo automática

En la Introducción, mencione que JavaScript tiende a salirse de su camino para aceptar casi cualquier programa que le demos, incluso programas que hacen cosas extrañas. Esto es bien demostrado por las siguientes expresiones:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("cinco" * 2)
// → NaN
console.log(false == 0)
// → true
```

Cuando un operador es aplicado al tipo de valor "incorrecto", JavaScript silenciosamente convertirá ese valor al tipo que necesita, utilizando una serie de reglas que frecuentemente no dan el resultado que quisieras o esperarías. Esto es llamado *coercion de tipo*. El `null` en la primera expresión se torna `0`, y el `"5"`en la segunda expresión se torna `5` (de string a número). Sin embargo, en la tercera expresión, `+` intenta realizar una concatenación de string antes que una adición numérica, entonces el `1` es convertido a `"1"` (de número a string)

Cuando algo que no se traduce a un número en una manera obvia (tal como `"cinco"` o `undefined`) es convertido a un número, obtenemos el valor `NaN`. Operaciones aritméticas subsecuentes con `NaN`, continúan produciendo `NaN`, asi

que si te encuentras obteniendo uno de estos valores en algun lugar inesperado, busca por coerciones de tipo accidentales.

Cuando se utiliza `==` para comparar valores del mismo tipo, el desenlace es fácil de predecir: debemos de obtener verdadero cuando ambos valores son lo mismo, excepto en el caso de `NaN`. Pero cuando los tipos difieren, JavaScript utiliza una serie de reglas complicadas y confusas para determinar que hacer. En la mayoria de los casos, solo tratara de convertir uno de estos valores al tipo del otro valor. Sin embargo, cuando `null` o `undefined` ocurren en cualquiera de los lados del operador, este produce verdadero solo si ambos lados son valores o `null` o `undefined`.

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

Este comportamiento es frecuentemente util. Cuando queremos probar si un valor tiene un valor real en vez de `null` o `undefined`, puedes compararlo con `null` usando el operador `==` (o `!=`).

Pero que pasa si queremos probar que algo se refiere precisamente al valor `false`? Las reglas para convertir strings y números a valores Booleanos, dice que `0`, `NaN`, y el string vació (`""`) cuentan como `false`, mientras que todos los otros valores cuentan como `true`. Debido a esto, expresiones como `0 == false`, y `"" == false` son también verdaderas. Cuando no queremos ninguna conversion de tipo automática, existen otros dos operadores adicionales: `===` y `!==`. El primero prueba si un valor es *precisamente* igual al otro, y el segundo prueba si un valor no es precisamente igual. Entonces `"" === false` es falso, como es de esperarse.

Recomiendo usar el operador de comparación de tres caracteres de una manera defensiva para prevenir que conversiones de tipo inesperadas te estorben. Pero cuando estés seguro de que el tipo va a ser el mismo en ambos lados, no es problemático utilizar los operadores mas cortos.

Corto circuito de operadores lógicos

Los operadores lógicos `&&` y `||`, manejan valores de diferentes tipos de una forma peculiar. Ellos convertirán el valor en su lado izquierdo a un tipo Booleano para decidir que hacer, pero dependiendo del operador y el resultado de la conversión, devolverán o el valor *original* de la izquierda o el valor de la derecha.

El operador `||`, por ejemplo, devolverá el valor de su izquierda cuando este

puede ser convertido a verdadero y de ser lo contrario devolverá el valor de la derecha. Esto tiene el efecto esperado cuando los valores son Booleanos, pero se comporta de una forma algo análoga con valores de otros tipos.

```
console.log(null || "usuario")
// → usuario
console.log("Agnes" || "usuario")
// → Agnes
```

Podemos utilizar esta funcionalidad como una forma de recurrir a un valor por defecto. Si tenemos un valor que puede estar vacío, podemos usar `||` después de este para remplazarlo con otro valor. Si el valor inicial puede ser convertido a falso, obtendra el reemplazo en su lugar.

El operador `&&` funciona de manera similar, pero de forma opuesta. Cuando el valor a su izquierda es algo que se convierte a falso, devuelve ese valor, y de lo contrario, devuelve el valor a su derecha.

Otra propiedad importante de estos dos operadores es que la parte de su derecha solo es evaluada si es necesario. En el caso de de `true || X`, no importa que sea `X`—aun si es una pieza del programa que hace algo *terrible*—el resultado será verdadero, y `X` nunca sera evaluado. Lo mismo sucede con `false && X`, que es falso e ignorará `X`. Esto es llamado *evaluación de corto circuito*.

El operador condicional funciona de manera similar. Del segundo y tercer valor, solo el que es seleccionado es evaluado.

Resumen

Observamos cuatro tipos de valores de JavaScript en este capítulo: números, textos (`strings`), Booleanos, y valores indefinidos.

Tales valores son creados escribiendo su nombre (`true`, `null`) o valor (`13`, `"abc"`). Puedes combinar y transformar valores con operadores. Vimos operadores binarios para aritmética (`+`, `-`, `*`, `/`, y `%`), concatenación de strings (`+`), comparaciones (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`), y lógica (`&&`, `||`), así también como varios otros operadores unarios (`-` para negar un número, `!` para negar lógicamente, y `typeof` para saber el valor de un tipo) y un operador ternario (`?:`) para elegir uno de dos valores basándose en un tercer valor.

Esto te dá la información suficiente para usar JavaScript como una calculadora de bolsillo, pero no para mucho más. El próximo capitulo empezará a juntar estas expresiones para formar programas básicos.

> *"Y mi corazón brilla de un color rojo brillante bajo mi piel transparente y translúcida, y tienen que administrarme 10cc de JavaScript para conseguir que regrese. (respondo bien a las toxinas en la sangre.) Hombre, esa cosa es increible!"*
>
> —_why, Why's (Poignant) Guide to Ruby

Chapter 2

Estructura de Programa

En este capítulo, comenzaremos a hacer cosas que realmente se pueden llamar *programación*. Expandiremos nuestro dominio del lenguaje JavaScript más allá de los sustantivos y fragmentos de oraciones que hemos visto hasta ahora, al punto donde podemos expresar prosa significativa.

Expresiones y declaraciones

En el Capítulo 1, creamos valores y les aplicamos operadores a ellos para obtener nuevos valores. Crear valores de esta manera es la sustancia principal de cualquier programa en JavaScript. Pero esa sustancia tiene que enmarcarse en una estructura más grande para poder ser útil. Así que eso es lo que veremos a continuación.

Un fragmento de código que produce un valor se llama una *expresión*. Cada valor que se escribe literalmente (como `22` o `"psicoanálisis"`) es una expresión. Una expresión entre paréntesis también es una expresión, como lo es un operador binario aplicado a dos expresiones o un operador unario aplicado a una sola.

Esto demuestra parte de la belleza de una interfaz basada en un lenguaje. Las expresiones pueden contener otras expresiones de una manera muy similar a como las sub-oraciones en los lenguajes humanos están anidadas, una sub-oración puede contener sus propias sub-oraciones, y así sucesivamente. Esto nos permite construir expresiones que describen cálculos arbitrariamente complejos.

Si una expresión corresponde al fragmento de una oración, una *declaración* en JavaScript corresponde a una oración completa. Un programa es una lista de declaraciones.

El tipo más simple de declaración es una expresión con un punto y coma después ella. Esto es un programa:

```
1;
!false;
```

Sin embargo, es un programa inútil. Una expresión puede estar feliz solo con producir un valor, que luego pueda ser utilizado por el código circundante. Una declaración es independiente por si misma, por lo que equivale a algo solo si afecta al mundo. Podría mostrar algo en la pantalla—eso cuenta como cambiar el mundo—o podría cambiar el estado interno de la máquina en una manera que afectará a las declaraciones que vengan después de ella. Estos cambios se llaman *efecto secundarios*. Las declaraciones en el ejemplo anterior solo producen los valores `1` y `true` y luego inmediatamente los tira a la basura. Esto no deja ninguna huella en el mundo. Cuando ejecutes este programa, nada observable ocurre.

En algunos casos, JavaScript te permite omitir el punto y coma al final de una declaración. En otros casos, tiene que estar allí, o la próxima línea serán tratada como parte de la misma declaración. Las reglas para saber cuando se puede omitir con seguridad son algo complejas y propensas a errores. Asi que en este libro, cada declaración que necesite un punto y coma siempre tendra uno. Te recomiendo que hagas lo mismo, al menos hasta que hayas aprendido más sobre las sutilezas de los puntos y comas que puedan ser omitidos.

VINCULACIONES

Cómo mantiene un programa un estado interno? Cómo recuerda cosas? Hasta ahora hemos visto cómo producir nuevos valores a partir de valores anteriores, pero esto no cambia los valores anteriores, y el nuevo valor tiene que ser usado inmediatamente o se disipará nuevamente. Para atrapar y mantener valores, JavaScript proporciona una cosa llamada *vinculación*, o *variable*:

```
let atrapado = 5 * 5;
```

Ese es un segundo tipo de declaración. La palabra especial (*palabra clave*) `let` indica que esta oración va a definir una vinculación. Le sigue el nombre de la vinculación y, si queremos darle un valor inmediatamente, un operador `=` y una expresión.

La declaración anterior crea una vinculación llamada `atrapado` y la usa para capturar el número que se produce al multiplicar 5 por 5.

Después de que una vinculación haya sido definida, su nombre puede usarse como una expresión. El valor de tal expresión es el valor que la vinculación mantiene actualmente. Aquí hay un ejemplo:

```
let diez = 10;
```

```
console.log(diez * diez);
// → 100
```

Cuando una vinculación señala a un valor, eso no significa que esté atada a ese valor para siempre. El operador `=` puede usarse en cualquier momento en vinculaciones existentes para desconectarlas de su valor actual y hacer que ellas apuntan a uno nuevo:

```
let humor = "ligero";
console.log(humor);
// → ligero
humor = "oscuro";
console.log(humor);
// → oscuro
```

Deberías imaginar a las vinculaciones como tentáculos, en lugar de cajas. Ellas no *contienen* valores; ellas los *agarran*—dos vinculaciones pueden referirse al mismo valor. Un programa solo puede acceder a los valores que todavía pueda referenciar. Cuando necesitas recordar algo, creces un tentáculo para aferrarte a él o vuelves a conectar uno de tus tentáculos existentes a ese algo.

Veamos otro ejemplo. Para recordar la cantidad de dólares que Luigi aún te debe, creas una vinculación. Y luego, cuando él te pague de vuelta $35, le das a esta vinculación un nuevo valor:

```
let deudaLuigi = 140;
deudaLuigi = deudaLuigi - 35;
console.log(deudaLuigi);
// → 105
```

Cuando defines una vinculación sin darle un valor, el tentáculo no tiene nada que agarrar, por lo que termina en solo aire. Si pides el valor de una vinculación vacía, obtendrás el valor `undefined`.

Una sola declaración `let` puede definir múltiples vinculaciones. Las definiciones deben estar separadas por comas.

```
let uno = 1, dos = 2;
console.log(uno + dos);
// → 3
```

Las palabras `var` y `const` también pueden ser usadas para crear vinculaciones,

en una manera similar a `let`.

```
var nombre = "Ayda";
const saludo = "Hola ";
console.log(saludo + nombre);
// → Hola Ayda
```

La primera, `var` (abreviatura de "variable"), es la forma en la que se declaraban las vinculaciones en JavaScript previo al 2015. Volveremos a la forma precisa en que difiere de `let` en el próximo capítulo. Por ahora, recuerda que generalmente hace lo mismo, pero raramente la usaremos en este libro porque tiene algunas propiedades confusas.

La palabra `const` representa una *constante*. Define una vinculación constante, que apunta al mismo valor por el tiempo que viva. Esto es útil para vinculaciones que le dan un nombre a un valor para que fácilmente puedas consultarlo más adelante.

Nombres vinculantes

Los nombres de las vinculaciones pueden ser cualquier palabra. Los dígitos pueden ser parte de los nombres de las vinculaciones pueden—`catch22` es un nombre válido, por ejemplo—pero el nombre no debe comenzar con un dígito. El nombre de una vinculación puede incluir signos de dólar (`$`) o caracteres de subrayado (`_`), pero no otros signos de puntuación o caracteres especiales.

Las palabras con un significado especial, como `let`, son *palabras claves*, y no pueden usarse como nombres vinculantes. También hay una cantidad de palabras que están "reservadas para su uso" en futuras versiones de JavaScript, que tampoco pueden ser usadas como nombres vinculantes. La lista completa de palabras clave y palabras reservadas es bastante larga:

```
break case catch class const continue debugger default
delete do else enum export extends false finally for
function if implements import interface in instanceof let
new package private protected public return static super
switch this throw true try typeof var void while with yield
```

No te preocupes por memorizarlas. Cuando crear una vinculación produzca un error de sintaxis inesperado, observa si estas tratando de definir una palabra reservada.

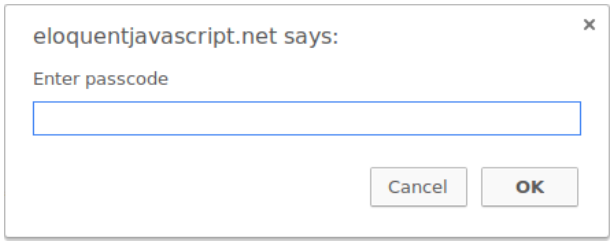
El entorno

La colección de vinculaciones y sus valores que existen en un momento dado se llama *entorno*. Cuando se inicia un programa, est entorno no está vacío. Siempre contiene vinculaciones que son parte del estándar del lenguaje, y la mayoría de las veces, también tiene vinculaciones que proporcionan formas de interactuar con el sistema circundante. Por ejemplo, en el navegador, hay funciones para interactuar con el sitio web actualmente cargado y para leer entradas del mouse y teclado.

Funciones

Muchos de los valores proporcionados por el entorno predeterminado tienen el tipo *función*. Una función es una pieza de programa envuelta en un valor. Dichos valores pueden ser *aplicados* para ejecutar el programa envuelto. Por ejemplo, en un entorno navegador, la vinculación `prompt` sostiene una función que muestra un pequeño cuadro de diálogo preguntando por entrada del usuario. Esta se usa así:

```
prompt("Introducir contraseña");
```



Ejecutar una función tambien se conoce como *invocarla*, *llamarla*, o *aplicarla*. Puedes llamar a una función poniendo paréntesis después de una expresión que produzca un valor de función. Usualmente usarás directamente el nombre de la vinculación que contenga la función. Los valores entre los paréntesis se dan al programa dentro de la función. En el ejemplo, la función `prompt` usa el string que le damos como el texto a mostrar en el cuadro de diálogo. Los valores dados a las funciones se llaman *argumentos*. Diferentes funciones pueden necesitar un número diferente o diferentes tipos de argumentos

La función `prompt` no se usa mucho en la programación web moderna, sobre todo porque no tienes control sobre la forma en como se ve la caja de diálogo resultante, pero puede ser útil en programas de juguete y experimentos.

La función console.log

En los ejemplos, utilicé `console.log` para dar salida a los valores. La mayoría de los sistemas de JavaScript (incluidos todos los navegadores web modernos y Node.js) proporcionan una función `console.log` que escribe sus argumentos en *algun* dispositivo de salida de texto. En los navegadores, esta salida aterriza en la consola de JavaScript. Esta parte de la interfaz del navegador está oculta por defecto, pero la mayoría de los navegadores la abren cuando presionas F12 o, en Mac, Command-Option-I. Si eso no funciona, busca en los menús un elemento llamado "herramientas de desarrollador" o algo similar.

Aunque los nombres de las vinculaciones no puedan contener carácteres de puntos, `console.log` tiene uno. Esto es porque `console.log` no es un vinculación simple. En realidad, es una expresión que obtiene la propiedad `log` del valor mantenido por la vinculación `console`. Averiguaremos qué significa esto exactamente en el Capítulo 4.

Valores de retorno

Mostrar un cuadro de diálogo o escribir texto en la pantalla es un *efecto secundario*. Muchas funciones son útiles debido a los efectos secundarios que ellas producen. Las funciones también pueden producir valores, en cuyo caso no necesitan tener un efecto secundario para ser útil. Por ejemplo, la función `Math.max` toma cualquier cantidad de argumentos numéricos y devuelve el mayor de ellos.

```
console.log(Math.max(2, 4));
// → 4
```

Cuando una función produce un valor, se dice que *retorna* ese valor. Todo lo que produce un valor es una expresión en JavaScript, lo que significa que las llamadas a funciones se pueden usar dentro de expresiones más grandes. aquí una llamada a `Math.min`, que es lo opuesto a `Math.max`, se usa como parte de una expresión de adición:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

El próximo capítulo explica cómo escribir tus propias funciones.

Flujo de control

Cuando tu programa contiene más de una declaración, las declaraciones se ejecutan como si fueran una historia, de arriba a abajo. Este programa de ejemplo tiene dos declaraciones. La primera le pide al usuario un número, y la segunda, que se ejecuta después de la primera, muestra el cuadrado de ese número.

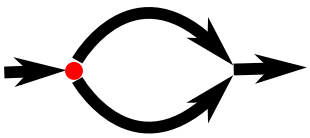
```
let elNumero = Number(prompt("Elige un numero"));
console.log("Tu número es la raiz cuadrada de " +
            elNumero * elNumero);
```

La función `Número` convierte un valor a un número. Necesitamos esa conversión porque el resultado de `prompt` es un valor de string, y nosotros queremos un numero. Hay funciones similares llamadas `String` y `Boolean` que convierten valores a esos tipos.

Aquí está la representación esquemática (bastante trivial) de un flujo de control en línea recta:

Ejecución condicional

No todos los programas son caminos rectos. Podemos, por ejemplo, querer crear un camino de ramificación, donde el programa toma la rama adecuada basadandose en la situación en cuestión. Esto se llama *ejecución condicional*.



La ejecución condicional se crea con la palabra clave `if` en JavaScript. En el caso simple, queremos que se ejecute algún código si, y solo si, una cierta condición se cumple. Podríamos, por ejemplo, solo querer mostrar el cuadrado de la entrada si la entrada es realmente un número.

```
let elNumero = Number(prompt("Elige un numero"));
if (!Number.isNaN(elNumero)) {
  console.log("Tu número es la raiz cuadrada de " +
              elNumero * elNumero);
}
```

Con esta modificación, si ingresas la palabra “loro”, no se mostrara ninguna salida.

La palabra clave `if` ejecuta u omite una declaración dependiendo del valor de una expresión booleana. La expresión decisiva se escribe después de la palabra clave, entre paréntesis, seguida de la declaración a ejecutar.

La función `Number.isNaN` es una función estándar de JavaScript que retorna `true` solo si el argumento que se le da es `NaN`. Resulta que la función `Number` devuelve `NaN` cuando le pasas un string que no representa un número válido. Por lo tanto, la condición se traduce a “a menos que `elNumero` no sea un número, haz esto”.

La declaración debajo del `if` está envuelta en llaves (`{`y `}`) en este ejemplo. Estos pueden usarse para agrupar cualquier cantidad de declaraciones en una sola declaración, llamada un *bloque*. Podrías también haberlas omitido en este caso, ya que solo tienes una sola declaración, pero para evitar tener que pensar si se necesitan o no, la mayoría de los programadores en JavaScript las usan en cada una de sus declaraciones envueltas como esta. Seguiremos esta convención en la mayoria de este libro, a excepción de la ocasional declaración de una sola linea.

```
if (1 + 1 == 2) console.log("Es verdad");
// → Es verdad
```

A menudo no solo tendrás código que se ejecuta cuando una condición es verdadera, pero también código que maneja el otro caso. Esta ruta alternativa está representado por la segunda flecha en el diagrama. La palabra clave `else` se puede usar, junto con `if`, para crear dos caminos de ejecución alternativos, de una manera separada.

```
let elNumero = Number(prompt("Elige un numero"));
if (!Number.isNaN(elNumero)) {
  console.log("Tu número es la raiz cuadrada de " +
              elNumero * elNumero);
} else {
  console.log("Ey. Por qué no me diste un número?");
}
```

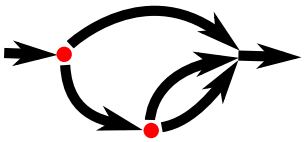
Si tenemos más de dos rutas a elegir, múltiples pares de `if`/`else` se pueden “encadenar”. Aquí hay un ejemplo:

```
let numero = Number(prompt("Elige un numero"));
```

```
if (numero < 10) {
  console.log("Pequeño");
} else if (numero < 100) {
  console.log("Mediano");
} else {
  console.log("Grande");
}
```

El programa primero comprobará si `numero` es menor que 10. Si lo es, eligira esa rama, mostrara `"Pequeño"`, y está listo. Si no es así, toma la rama `else`, que a su vez contiene un segundo `if`. Si la segunda condición (`< 100`) es verdadera, eso significa que el número está entre 10 y 100, y `"Mediano"` se muestra. Si no es así, la segunda y última la rama `else` es elegida.

El esquema de este programa se ve así:

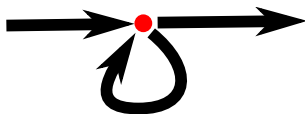


Ciclos while y do

Considera un programa que muestra todos los números pares de 0 a 12. Una forma de escribir esto es la siguiente:

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

Eso funciona, pero la idea de escribir un programa es hacer de algo *menos* trabajo, no más. Si necesitáramos todos los números pares menores a 1.000, este enfoque sería poco práctico. Lo que necesitamos es una forma de ejecutar una pieza de código multiples veces. Esta forma de flujo de control es llamada un *ciclo* (o “loop”):



El flujo de control de ciclos nos permite regresar a algún punto del programa en donde estábamos antes y repetirlo con nuestro estado del programa actual. Si combinamos esto con una vinculación que cuenta, podemos hacer algo como esta:

```
let numero = 0;
while (numero <= 12) {
  console.log(numero);
  numero = numero + 2;
}
// → 0
// → 2
// …   etcetera
```

Una declaración que comienza con la palabra clave `while` crea un ciclo. La palabra `while` es seguida por una expresión en paréntesis y luego por una declaración, muy similar a `if`. El bucle sigue ingresando a esta declaración siempre que la expresión produzca un valor que dé `true` cuando sea convertida a Boolean.

La vinculación `numero` demuestra la forma en que una vinculaciónpuede seguir el progreso de un programa. Cada vez que el ciclo se repite, `numero` obtiene un valor que es 2 más que su valor anterior. Al comienzo de cada repetición, se compara con el número 12 para decidir si el trabajo del programa está terminado.

Como un ejemplo que realmente hace algo útil, ahora podemos escribir un programa que calcula y muestra el valor de 2^{10} (2 a la 10). Usamos dos vinculaciones: una para realizar un seguimiento de nuestro resultado y una para contar cuántas veces hemos multiplicado este resultado por 2. El ciclo prueba si la segunda vinculación ha llegado a 10 todavía y, si no, actualiza ambas vinculaciones.

```
let resultado = 1;
let contador = 0;
while (contador < 10) {
  resultado = resultado * 2;
  contador = contador + 1;
}
console.log(resultado);
// → 1024
```

El contador también podría haber comenzado en `1` y chequear para `<= 10`, pero, por razones que serán evidentes en el Capítulo 4, es una buena idea ir acostumbrandose a contar desde 0.

Un ciclo `do` es una estructura de control similar a un ciclo `while`. Difiere solo en un punto: un ciclo `do` siempre ejecuta su cuerpo al menos una vez, y comienza a chequear si debe detenerse solo después de esa primera ejecución. Para reflejar esto, la prueba aparece después del cuerpo del ciclo:

```
let tuNombre;
do {
  tuNombre = prompt("Quien eres?");
} while (!tuNombre);
console.log(tuNombre);
```

Este programa te obligará a ingresar un nombre. Preguntará de nuevo y de nuevo hasta que obtenga algo que no sea un string vacío. Aplicar el operador `!` convertirá un valor a tipo Booleano antes de negarlo y todos los strings, excepto `""` seran convertidas a `true`. Esto significa que el ciclo continúa dando vueltas hasta que proporciones un nombre no-vacío.

Indentando Código

En los ejemplos, he estado agregando espacios adelante de declaraciones que son parte de una declaración más grande. Estos no son necesarios—la computadora aceptará el programa normalmente sin ellos. De hecho, incluso las nuevas líneas en los programas son opcionales. Podrías escribir un programa en una sola línea inmensa si asi quisieras.

El rol de esta indentación dentro de los bloques es hacer que la estructura del código se destaque. En código donde se abren nuevos bloques dentro de otros bloques, puede ser difícil ver dónde termina un bloque y donde comienza el otro. Con la indentación apropiada, la forma visual de un programa corresponde a la forma de los bloques dentro de él. Me gusta usar dos espacios para cada bloque abierto, pero los gustos varían—algunas personas usan cuatro espacios, y algunas personas usan carácteres de tabulación. Lo cosa importante es que cada bloque nuevo agregue la misma cantidad de espacio.

```
if (false != true) {
  console.log("Esto tiene sentido.");
  if (1 < 2) {
    console.log("Ninguna sorpresa alli.");
```

```
  }
}
```

La mayoría de los editores de código ayudaran indentar automáticamente las nuevas líneas con la cantidad adecuada.

Ciclos for

Muchos ciclos siguen el patrón visto en los ejemplos de `while`. Primero una vinculación "contador" se crea para seguir el progreso del ciclo. Entonces viene un ciclo `while`, generalmente con una expresión de prueba que verifica si el contador ha alcanzado su valor final. Al final del cuerpo del ciclo, el el contador se actualiza para mantener un seguimiento del progreso.

Debido a que este patrón es muy común, JavaScript y otros lenguajes similares proporcionan una forma un poco más corta y más completa, el ciclo `for`:

```
for (let numero = 0; numero <= 12; numero = numero + 2) {
  console.log(numero);
}
// → 0
// → 2
// …   etcetera
```

Este programa es exactamente equivalente al ejemplo anterior de impresión de números pares. El único cambio es que todos las declaraciónes que están relacionadas con el "estado" del ciclo estan agrupadas después del `for`.

Los paréntesis después de una palabra clave `for` deben contener dos punto y comas. La parte antes del primer punto y coma *inicializa* el cicloe, generalmente definiendo una vinculación. La segunda parte es la expresión que *chequea* si el ciclo debe continuar. La parte final *actualiza* el estado del ciclo después de cada iteración. En la mayoría de los casos, esto es más corto y conciso que un constructo `while`.

Este es el código que calcula 2^{10}, usando `for` en lugar de `while`:

```
let resultado = 1;
for (let contador = 0; contador < 10; contador = contador + 1) {
  resultado = resultado * 2;
}
console.log(resultado);
// → 1024
```

Rompiendo un ciclo

Hacer que la condición del ciclo produzca `false` no es la única forma en que el ciclo puede terminar. Hay una declaración especial llamada `break` ("romper") que tiene el efecto de inmediatamente saltar afuera del ciclo circundante.

Este programa ilustra la declaración `break`. Encuentra el primer número que es a la vez mayor o igual a 20 y divisible por 7.

```
for (let actual = 20; ; actual = actual + 1) {
  if (actual % 7 == 0) {
    console.log(actual);
    break;
  }
}
// → 21
```

Usar el operador restante (`%`) es una manera fácil de probar si un número es divisible por otro número. Si lo es, el residuo de su división es cero.

El constructo `for` en el ejemplo no tiene una parte que verifique cuando finalizar el ciclo. Esto significa que el ciclo nunca se detendrá a menos que se ejecute la declaración `break` dentro de el.

Si eliminases esa declaración `break` o escribieras accidentalmente una condición final que siempre produciera `true`, tu programa estaria atrapado en un *ciclo infinito*. Un programa atrapado en un ciclo infinito nunca terminará de ejecutarse, lo que generalmente es algo malo.

La palabra clave `continue` ("continuar") es similar a `break`, en que influye el progreso de un ciclo. Cuando `continue` se encuentre en el cuerpo de un ciclo, el control salta afuera del cuerpo y continúa con la siguiente iteración del ciclo.

Actualizando vinculaciones de manera sucinta

Especialmente cuando realices un ciclo, un programa a menudo necesita "actualizar" una vinculación para mantener un valor basadandose en el valor anterior de esa vinculación.

```
contador = contador + 1;
```

JavaScript provee de un atajo para esto:

```
contador += 1;
```

Atajos similares funcionan para muchos otros operadores, como `resultado *= 2` para duplicar `resultado` o `contador -= 1` para contar hacia abajo.

Esto nos permite acortar un poco más nuestro ejemplo de conteo.

```
for (let numero = 0; numero <= 12; numero += 2) {
  console.log(numero);
}
```

Para `contador += 1` y `contador -= 1`, hay incluso equivalentes más cortos: `contador++` y `contador --`.

Despachar en un valor con switch

No es poco común que el código se vea así:

```
if (x == "valor1") accion1();
else if (x == "valor2") accion2();
else if (x == "valor3") accion3();
else accionPorDefault();
```

Existe un constructo llamado `switch` que está destinada a expresar tales "despachos" de una manera más directa. Desafortunadamente, la sintaxis que JavaScript usa para esto (que heredó de la línea lenguajes de programación C/Java) es algo incómoda—una cadena de declaraciones `if` podria llegar a verse mejor. Aquí hay un ejemplo:

```
switch (prompt("Como esta el clima?")) {
  case "lluvioso":
    console.log("Recuerda salir con un paraguas.");
    break;
  case "soleado":
    console.log("Vistete con poca ropa.");
  case "nublado":
    console.log("Ve afuera.");
    break;
  default:
    console.log("Tipo de clima desconocido!");
    break;
}
```

Puedes poner cualquier número de etiquetas de `case` dentro del bloque abierto por `switch`. El programa comenzará a ejecutarse en la etiqueta que corresponde al valor que se le dio a `switch`, o en `default` si no se encuentra ningún valor que coincida. Continuará ejecutándose, incluso a través de otras etiquetas, hasta que llegue a una declaración `break`. En algunos casos, como en el caso `"soleado"` del ejemplo, esto se puede usar para compartir algo de código entre casos (recomienda salir para ambos climas soleado y nublado). Pero ten cuidado—es fácil olvidarse de `break`, lo que hará que el programa ejecute código que no quieres que sea ejecutado.

Capitalización

Los nombres de vinculaciones no pueden contener espacios, sin embargo, a menudo es útil usar múltiples palabras para describir claramente lo que representa la vinculación. Estas son más o menos tus opciones para escribir el nombre de una vinculación con varias palabras en ella:

```
pequeñatortugaverde
pequeña_tortuga_verde
PequeñaTortugaVerde
pequeñaTortugaVerde
```

El primer estilo puede ser difícil de leer. Me gusta mucho el aspecto del estilo con los guiones bajos, aunque ese estilo es algo fastidioso de escribir. Las funciones estándar de JavaScript, y la mayoría de los programadores de JavaScript, siguen el estilo de abajo: capitalizan cada palabra excepto la primera. No es difícil acostumbrarse a pequeñas cosas así, y programar con estilos de nombres mixtos pueden ser algo discordante para leer, así que seguiremos esta convención.

En algunos casos, como en la función `Number`, la primera letra de la vinculación también está en mayúscula. Esto se hizo para marcar esta función como un constructor. Lo que es un constructor quedará claro en el Capítulo 6. Por ahora, lo importante es no ser molestado por esta aparente falta de consistencia.

Comentarios

A menudo, el código en si mismo no transmite toda la información que deseas que un programa transmita a los lectores humanos, o lo transmite de una manera tan críptica que la gente quizás no lo entienda. En otras ocasiones, podrías simplemente querer incluir algunos pensamientos relacionados como parte de tu programa. Esto es para lo qué son los *comentarios*.

Un comentario es una pieza de texto que es parte de un programa pero que es completamente ignorado por la computadora. JavaScript tiene dos formas de escribir comentarios. Para escribir un comentario de una sola línea, puede usar dos caracteres de barras inclinadas (`//`) y luego el texto del comentario después.

```
let balanceDeCuenta = calcularBalance(cuenta);
// Es un claro del bosque donde canta un río
balanceDeCuenta.ajustar();
// Cuelgan enloquecidamente de las hierbas harapos de plata
let reporte = new Reporte();
// Donde el sol de la orgullosa montaña luce:
añadirAReporte(balanceDeCuenta, reporte);
// Un pequeño valle espumoso de luz.
```

Un comentario `//` va solo haste el final de la línea. Una sección de texto entre `/*` y `*/` se ignorará en su totalidad, independientemente de si contiene saltos de línea. Esto es útil para agregar bloques de información sobre un archivo o un pedazo de programa.

```
/*
  Primero encontré este número garabateado en la parte posterior de
  un viejo cuaderno. Desde entonces, a menudo lo he visto,
  apareciendo en números de teléfono y en los números de serie de
  productos que he comprado. Obviamente me gusta, así que
  decidí quedármelo
*/
const miNumero = 11213;
```

Resumen

Ahora sabes que un programa está construido a partir de declaraciones, las cuales a veces pueden contener más declaraciones. Las declaraciones tienden a

contener expresiones, que a su vez se pueden construir a partir de expresiones mas pequeñas.

Poner declaraciones una despues de otras te da un programa que es ejecutado de arriba hacia abajo. Puedes introducir alteraciones en el flujo de control usando declaraciones condicionales (`if`, `else`, y `switch`) y ciclos (`while`, `do`, y `for`).

Las vinculaciones se pueden usar para archivar datos bajo un nombre, y son utiles para el seguimiento de estado en tu programa. El entorno es el conjunto de vinculaciones que se definen. Los sistemas de JavaScript siempre incluyen por defecto un número de vinculaciones estándar útiles en tu entorno.

Las funciones son valores especiales que encapsulan una parte del programa. Puedes invocarlas escribiendo `nombreDeLaFuncion(argumento1, argumento2)`. Tal llamada a función es una expresión, y puede producir un valor.

Ejercicios

Si no estas seguro de cómo probar tus soluciones para los ejercicios, consulta la introducción.

Cada ejercicio comienza con una descripción del problema. Lee eso y trata de resolver el ejercicio. Si tienes problemas, considera leer las pistas en el final del libro. Las soluciones completas para los ejercicios no estan incluidas en este libro, pero puedes encontrarlas en línea en *eloquentjavascript.net/code*. Si quieres aprender algo de los ejercicios, te recomiendo mirar a las soluciones solo despues de que hayas resuelto el ejercicio, o al menos despues de que lo hayas intentando resolver por un largo tiempo y tengas un ligero dolor de cabeza.

Ciclo de un triángulo

Escriba un ciclo que haga siete llamadas a `console.log` para generar el siguiente triángulo:

```
#
##
###
####
#####
######
#######
```

Puede ser útil saber que puedes encontrar la longitud de un string escribiendo `.length` después de él:

```
let abc = "abc";
console.log(abc.length);
// → 3
```

FizzBuzz

Escribe un programa que use `console.log` para imprimir todos los números de 1 a 100, con dos excepciones. Para números divisibles por 3, imprime `"Fizz"` en lugar del número, y para los números divisibles por 5 (y no 3), imprime `"Buzz"` en su lugar.

Cuando tengas eso funcionando, modifica tu programa para imprimir `"FizzBuzz"`, para números que sean divisibles entre 3 y 5 (y aún imprimir `"Fizz"` o `"Buzz"` para números divisibles por solo uno de ellos).

(Esta es en realidad una pregunta de entrevista que se ha dicho elimina un porcentaje significativo de candidatos a programadores. Así que si la puedes resolver, tu valor en el mercado laboral acaba de subir).

Tablero de ajedrez

Escribe un programa que cree un string que represente una cuadrícula de 8×8, usando caracteres de nueva línea para separar las líneas. En cada posición de la cuadrícula hay un espacio o un carácter "#". Los caracteres deberían de formar un tablero de ajedrez.

Pasar este string a `console.log` debería mostrar algo como esto:

```
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
```

Cuando tengas un programa que genere este patrón, define una vinculación `tamaño = 8` y cambia el programa para que funcione con cualquier `tamaño`, dando como salida una cuadrícula con el alto y ancho dados.

"La gente piensa que las ciencias de la computación son el arte de los genios, pero la verdadera realidad es lo opuesto, estas solo consisten en mucha gente haciendo cosas que se construyen una sobre la otra, al igual que un muro hecho de piedras pequeñas."

—Donald Knuth

Chapter 3

Funciones

Las funciones son el pan y la mantequilla de la programación en JavaScript. El concepto de envolver una pieza de programa en un valor tiene muchos usos. Esto nos da una forma de estructurar programas más grandes, de reducir la repetición, de asociar nombres con subprogramas y de aislar estos subprogramas unos con otros.

La aplicación más obvia de las funciones es definir nuevo vocabulario. Crear nuevas palabras en la prosa suele ser un mal estilo. Pero en la programación, es indispensable.

En promedio, un tipico adulto que hable español tiene unas 20,000 palabras en su vocabulario. Pocos lenguajes de programación vienen con 20,000 comandos ya incorporados en el. Y el vocabulario que *está* disponible tiende a ser más precisamente definido, y por lo tanto menos flexible, que en el lenguaje humano. Por lo tanto, nosotros por lo general *tenemos* que introducir nuevos conceptos para evitar repetirnos demasiado.

Definiendo una función

Una definición de función es una vinculación regular donde el valor de la vinculación es una función. Por ejemplo, este código define `cuadrado` para referirse a una función que produce el cuadrado de un número dado:

```
const cuadrado = function(x) {
  return x * x;
};

console.log(cuadrado(12));
// → 144
```

Una función es creada con una expresión que comienza con la palabra clave `function` ("función"). Las funciones tienen un conjunto de *parámetros* (en este caso, solo `x`) y un *cuerpo*, que contiene las declaraciones que deben ser

ejecutadas cuando se llame a la función. El cuerpo de la función de una función creada de esta manera siempre debe estar envuelto en llaves, incluso cuando consista en una sola declaración.

Una función puede tener múltiples parámetros o ningún parámetro en absoluto. En el siguiente ejemplo, `hacerSonido` no lista ningún nombre de parámetro, mientras que `potencia` enumera dos:

```
const hacerSonido = function() {
  console.log("Pling!");
};

hacerSonido();
// → Pling!

const potencia = function(base, exponente) {
  let resultado = 1;
  for (let cuenta = 0; cuenta < exponente; cuenta++) {
    resultado *= base;
  }
  return resultado;
};

console.log(potencia(2, 10));
// → 1024
```

Algunas funciones producen un valor, como `potencia` y `cuadrado`, y algunas no, como `hacerSonido`, cuyo único resultado es un efecto secundario. Una declaración de `return` determina el valor que es retornado por la función. Cuando el control se encuentre con tal declaración, inmediatamente salta de la función actual y devuelve el valor retornado al código que llamó la función. Una declaración `return` sin una expresión después de ella hace que la función retorne `undefined`. Funciones que no tienen una declaración `return` en absoluto, como `hacerSonido`, similarmente retornan `undefined`.

Los parámetros de una función se comportan como vinculaciones regulares, pero sus valores iniciales están dados por el *llamador* de la función, no por el código en la función en sí.

Vinculaciones y alcances

Cada vinculación tiene un *alcace*, que correspone a la parte del programa en donde la vinculación es visible. Para vinculaciones definidas fuera de cualquier

función o bloque, el alcance es todo el programa—puedes referir a estas vinculaciones en donde sea que quieras. Estas son llamadas *globales*.

Pero las vinculaciones creadas como parámetros de función o declaradas dentro de una función solo puede ser referenciadas en esa función. Estas se llaman *locales*. Cada vez que se llame a la función, se crean nuevas instancias de estas vinculaciones. Esto proporciona cierto aislamiento entre funciones—cada llamada de función actúa sobre su pequeño propio mundo (su entorno local), y a menudo puede ser entendida sin saber mucho acerca de lo qué está pasando en el entorno global.

Vinculaciones declaradas con `let` y `const` son, de hecho, locales al *bloque* donde esten declarados, así que si creas uno de esas dentro de un ciclo, el código antes y después del ciclo no puede "verlas". En JavaScript anterior a 2015, solo las funciones creaban nuevos alcances, por lo que las vinculaciones de estilo-antiguo, creadas con la palabra clave `var`, son visibles a lo largo de toda la función en la que aparecen—o en todo el alcance global, si no están dentro de una función.

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
  // → 60
}
// y no es visible desde aqui
console.log(x + z);
// → 40
```

Cada alcance puede "mirar afuera" hacia al alcance que lo rodee, por lo que `x` es visible dentro del bloque en el ejemplo. La excepción es cuando vinculaciones múltiples tienen el mismo nombre—en ese caso, el código solo puede ver a la vinculación más interna. Por ejemplo, cuando el código dentro de la función `dividirEnDos` se refiera a `numero`, estara viendo su *propio* `numero`, no el `numero` en el alcance global.

```
const dividirEnDos = function(numero) {
  return numero / 2;
};

let numero = 10;
console.log(dividirEnDos(100));
// → 50
```

```
console.log(numero);
// → 10
```

Alcance anidado

JavaScript no solo distingue entre vinculaciones *globales* y *locales*. Bloques y funciones pueden ser creados dentro de otros bloques y funciones, produciendo múltiples grados de localidad.

Por ejemplo, esta función—que muestra los ingredientes necesarios para hacer un lote de humus—tiene otra función dentro de ella:

```
const humus = function(factor) {
  const ingrediente = function(cantidad, unidad, nombre) {
    let cantidadIngrediente = cantidad * factor;
    if (cantidadIngrediente > 1) {
      unidad += "s";
    }
    console.log(`${cantidadIngrediente} ${unidad} ${nombre}`);
  };
  ingrediente(1, "lata", "garbanzos");
  ingrediente(0.25, "taza", "tahini");
  ingrediente(0.25, "taza", "jugo de limón");
  ingrediente(1, "clavo", "ajo");
  ingrediente(2, "cucharada", "aceite de oliva");
  ingrediente(0.5, "cucharadita", "comino");
};
```

El código dentro de la función `ingrediente` puede ver la vinculación `factor` de la función externa. Pero sus vinculaciones locales, como `unidad` o `cantidadIngrediente`, no son visibles para la función externa.

En resumen, cada alcance local puede ver también todos los alcances locales que lo contengan. El conjunto de vinculaciones visibles dentro de un bloque está determinado por el lugar de ese bloque en el texto del programa. Cada alcance local puede tambien ver todos los alcances locales que lo contengan, y todos los alcances pueden ver el alcance global. Este enfoque para la visibilidad de vinculaciones es llamado *alcance léxico*.

Funciones como valores

Las vinculaciones de función simplemente actúan como nombres para una pieza específica del programa. Tal vinculación se define una vez y nunca cambia. Esto hace que sea fácil confundir la función con su nombre.

Pero los dos son diferentes. Un valor de función puede hacer todas las cosas que otros valores pueden hacer—puedes usarlo en expresiones arbitrarias, no solo llamarlo. Es posible almacenar un valor de función en una nueva vinculación, pasarla como argumento a una función, y así sucesivamente. Del mismo modo, una vinculación que contenga una función sigue siendo solo una vinculación regular y se le puede asignar un nuevo valor, asi:

```
let lanzarMisiles = function() {
  sistemaDeMisiles.lanzar("ahora");
};
if (modoSeguro) {
  lanzarMisiles = function() {/* no hacer nada */};
}
```

En el Capitulo 5, discutiremos las cosas interesantes que se pueden hacer al pasar valores de función a otras funciones.

Notación de declaración

Hay una forma ligeramente más corta de crear una vinculación de función. Cuando la palabra clave `function` es usada al comienzo de una declaración, funciona de una manera diferente.

```
function cuadrado(x) {
  return x * x;
}
```

Esta es una *declaración* de función. La declaración define la vinculación `cuadrado` y la apunta a la función dada. Esto es un poco mas facil de escribir, y no requiere un punto y coma después de la función.

Hay una sutileza con esta forma de definir una función.

```
console.log("El futuro dice:", futuro());

function futuro() {
  return "Nunca tendran autos voladores";
```

```
}
```

Este código funciona, aunque la función esté definida *debajo* del código que lo usa. Las declaraciones de funciones no son parte del flujo de control regular de arriba hacia abajo. Estas son conceptualmente trasladadas a la cima de su alcance y pueden ser utilizadas por todo el código en ese alcance. Esto es a veces útil porque nos da la libertad de ordenar el código en una forma que nos parezca significativa, sin preocuparnos por tener que definir todas las funciones antes de que sean utilizadas.

Funciones de flecha

Existe una tercera notación para funciones, que se ve muy diferente de las otras. En lugar de la palabra clave `function`, usa una flecha (`=>`) compuesta de los caracteres igual y mayor que (no debe ser confundida con el operador igual o mayor que, que se escribe `>=`).

```
const potencia = (base, exponente) => {
  let resultado = 1;
  for (let cuenta = 0; cuenta < exponente; cuenta++) {
    resultado *= base;
  }
  return resultado;
};
```

La flecha viene *después* de la lista de parámetros, y es seguida por el cuerpo de la función. Expresa algo así como "esta entrada (los parámetros) produce este resultado (el cuerpo)".

Cuando solo haya un solo nombre de parámetro, los paréntesis alrededor de la lista de parámetros pueden ser omitidos. Si el cuerpo es una sola expresión, en lugar de un bloque en llaves, esa expresión será retornada por parte de la función. Asi que estas dos definiciones de `cuadrado` hacen la misma cosa:

```
const cuadrado1 = (x) => { return x * x; };
const cuadrado2 = x => x * x;
```

Cuando una función de flecha no tiene parámetros, su lista de parámetros es solo un conjunto vacío de paréntesis.

```
const bocina = () => {
```

```
  console.log("Toot");
};
```

No hay una buena razón para tener ambas funciones de flecha y expresiones `function` en el lenguaje. Aparte de un detalle menor, que discutiremos en Capítulo 6, estas hacen lo mismo. Las funciones de flecha se agregaron en 2015, principalmente para que fuera posible escribir pequeñas expresiones de funciones de una manera menos verbosa. Las usaremos mucho en el Capitulo 5.

La pila de llamadas

La forma en que el control fluye a través de las funciones es algo complicado. Vamos a écharle un vistazo más de cerca. Aquí hay un simple programa que hace unas cuantas llamadas de función:

```
function saludar(quien) {
  console.log("Hola " + quien);
}
saludar("Harry");
console.log("Adios");
```

Un recorrido por este programa es más o menos así: la llamada a `saludar` hace que el control salte al inicio de esa función (línea 2). La función llama a `console.log`, la cual toma el control, hace su trabajo, y entonces retorna el control a la línea 2. Allí llega al final de la función `saludar`, por lo que vuelve al lugar que la llamó, que es la línea 4. La línea que sigue llama a `console.log` nuevamente. Después que esta función retorna, el programa llega a su fin.

Podríamos mostrar el flujo de control esquemáticamente de esta manera:

We could show the flow of control schematically like this:

```
no en una función
   en saludar
        en console.log
   en saludar
no en una función
   en console.log
no en una función
```

Ya que una función tiene que regresar al lugar donde fue llamada cuando esta

retorna, la computadora debe recordar el contexto de donde sucedió la llamada. En un caso, `console.log` tiene que volver a la función `saludar` cuando está lista. En el otro caso, vuelve al final del programa.

El lugar donde la computadora almacena este contexto es la *pila de llamadas*. Cada vez que se llama a una función, el contexto actual es almacenado en la parte superior de esta "pila". Cuando una función retorna, elimina el contexto superior de la pila y lo usa para continuar la ejecución.

Almacenar esta pila requiere espacio en la memoria de la computadora. Cuando la pila crece demasiado grande, la computadora fallará con un mensaje como "fuera de espacio de pila" o "demasiada recursividad". El siguiente código ilustra esto haciendo una pregunta realmente difícil a la computadora, que causara un ir y venir infinito entre las dos funciones. Mejor dicho, *sería* infinito, si la computadora tuviera una pila infinita. Como son las cosas, nos quedaremos sin espacio, o "explotaremos la pila".

```
function gallina() {
  return huevo();
}
function huevo() {
  return gallina();
}
console.log(gallina() + " vino primero.");
// → ??
```

Argumentos Opcionales

El siguiente código está permitido y se ejecuta sin ningún problema:

```
function cuadrado(x) { return x * x; }
console.log(cuadrado(4, true, "erizo"));
// → 16
```

Definimos `cuadrado` con solo un parámetro. Sin embargo, cuando lo llamamos con tres, el lenguaje no se queja. Este ignora los argumentos extra y calcula el cuadrado del primero.

JavaScript es de extremadamente mente-abierta sobre la cantidad de argumentos que puedes pasar a una función. Si pasa demasiados, los adicionales son ignorados. Si pasas muy pocos, a los parámetros faltantes se les asigna el valor `undefined`.

La desventaja de esto es que es posible—incluso probable—que accidentalmente pases la cantidad incorrecta de argumentos a las funciones. Y nadie te dira nada acerca de eso.

La ventaja es que este comportamiento se puede usar para permitir que una función sea llamada con diferentes cantidades de argumentos. Por ejemplo, esta función `menos` intenta imitar al operador `-` actuando ya sea en uno o dos argumentos

```
function menos(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(menos(10));
// → -10
console.log(menos(10, 5));
// → 5
```

Si escribes un operador `=` después un parámetro, seguido de una expresión, el valor de esa expresión reemplazará al argumento cuando este no sea dado.

Por ejemplo, esta versión de `potencia` hace que su segundo argumento sea opcional. Si este no es proporcionado o si pasas el valor `undefined`, este se establecerá en dos y la función se comportará como `cuadrado`.

```
function potencia(base, exponente = 2) {
  let resultado = 1;
  for (let cuenta = 0; cuenta < exponente; cuenta++) {
    resultado *= base;
  }
  return resultado;
}

console.log(potencia(4));
// → 16
console.log(potencia(2, 6));
// → 64
```

En el próximo capítulo, veremos una forma en el que el cuerpo de una función puede obtener una lista de todos los argumentos que son pasados. Esto es útil porque hace posible que una función acepte cualquier cantidad de argumentos. Por ejemplo, `console.log` hace esto—muetra en la consola todos los valores que se le den.

```
console.log("C", "O", 2);
// → C O 2
```

Cierre

La capacidad de tratar a las funciones como valores, combinado con el hecho de que las vinculaciones locales se vuelven a crear cada vez que una sea función es llamada, trae a la luz una pregunta interesante. Qué sucede con las vinculaciones locales cuando la llamada de función que los creó ya no está activa?

El siguiente código muestra un ejemplo de esto. Define una función, `envolverValor` , que crea una vinculación local. Luego retorna una función que accede y devuelve esta vinculación local.

```
function envolverValor(n) {
  let local = n;
  return () => local;
}

let envolver1 = envolverValor(1);
let envolver2 = envolverValor(2);
console.log(envolver1());
// → 1
console.log(envolver2());
// → 2
```

Esto está permitido y funciona como es de esperar—ambas instancias de las vinculaciones todavía pueden ser accedidas. Esta situación es una buena demostración del hecho de que las vinculaciones locales se crean de nuevo para cada llamada, y que las diferentes llamadas no pueden pisotear las distintas vinculaciones locales entre sí.

Esta característica—poder hacer referencia a una instancia específica de una vinculación local en un alcance encerrado—se llama *cierre*. Una función que que hace referencia a vinculaciones de alcances locales alrededor de ella es llamada *un* cierre. Este comportamiento no solo te libera de tener que preocuparte por la duración de las vinculaciones pero también hace posible usar valores de funciones en algunas formas bastante creativas.

Con un ligero cambio, podemos convertir el ejemplo anterior en una forma de crear funciones que multipliquen por una cantidad arbitraria.

```
function multiplicador(factor) {
  return numero => numero * factor;
}

let duplicar = multiplicador(2);
console.log(duplicar(5));
// → 10
```

La vinculación explícita `local` del ejemplo `envolverValor` no es realmente necesaria ya que un parámetro es en sí misma una vinculación local.

Pensar en programas de esta manera requiere algo de práctica. Un buen modelo mental es pensar en los valores de función como que contienen tanto el código en su cuerpo tanto como el entorno en el que se crean. Cuando son llamadas, el cuerpo de la función ve su entorno original, no el entorno en el que se realiza la llamada.

En el ejemplo, se llama a `multiplicador` y esta crea un entorno en el que su parámetro `factor` está ligado a 2. El valor de función que retorna, el cual se almacena en `duplicar`, recuerda este entorno. Asi que cuando es es llamada, multiplica su argumento por 2.

Recursión

Está perfectamente bien que una función se llame a sí misma, siempre que no lo haga tanto que desborde la pila. Una función que se llama a si misma es llamada *recursiva*. La recursión permite que algunas funciones sean escritas en un estilo diferente. Mira, por ejemplo, esta implementación alternativa de `potencia`:

```
function potencia(base, exponente) {
  if (exponente == 0) {
    return 1;
  } else {
    return base * potencia(base, exponente - 1);
  }
}

console.log(potencia(2, 3));
// → 8
```

Esto es bastante parecido a la forma en la que los matemáticos definen la exponenciación y posiblemente describa el concepto más claramente que la

variante con el ciclo. La función se llama a si misma muchas veces con cada vez exponentes más pequeños para lograr la multiplicación repetida.

Pero esta implementación tiene un problema: en las implementaciones típicas de JavaScript, es aproximadamente 3 veces más lenta que la versión que usa un ciclo. Correr a través de un ciclo simple es generalmente más barato en terminos de memoria que llamar a una función multiples veces.

El dilema de velocidad versus elegancia es interesante. Puedes verlo como una especie de compromiso entre accesibilidad-humana y accesibilidad-maquina. Casi cualquier programa se puede hacer más rápido haciendolo más grande y complicado. El programador tiene que decidir acerca de cual es un equilibrio apropiado.

En el caso de la función `potencia`, la versión poco elegante (con el ciclo) sigue siendo bastante simple y fácil de leer. No tiene mucho sentido reemplazarla con la versión recursiva. A menudo, sin embargo, un programa trata con conceptos tan complejos que renunciar a un poco de eficiencia con el fin de hacer que el programa sea más sencillo es útil.

Preocuparse por la eficiencia puede ser una distracción. Es otro factor más que complica el diseño del programa, y cuando estás haciendo algo que ya es difícil, añadir algo más de lo que preocuparse puede ser paralizante.

Por lo tanto, siempre comienza escribiendo algo que sea correcto y fácil de comprender. Si te preocupa que sea demasiado lento—lo que generalmente no sucede, ya que la mayoría del código simplemente no se ejecuta con la suficiente frecuencia como para tomar cantidades significativas de tiempo—puedes medir luego y mejorar si es necesario.

La recursión no siempre es solo una alternativa ineficiente a los ciclos. Algunos problemas son realmente más fáciles de resolver con recursión que con ciclos. En la mayoría de los casos, estos son problemas que requieren explorar o procesar varias "ramas", cada una de las cuales podría ramificarse de nuevo en aún más ramas.

Considera este acertijo: comenzando desde el número 1 y repetidamente agregando 5 o multiplicando por 3, una cantidad infinita de números nuevos pueden ser producidos. ¿Cómo escribirías una función que, dado un número, intente encontrar una secuencia de tales adiciones y multiplicaciones que produzca ese número?

Por ejemplo, se puede llegar al número 13 multiplicando primero por 3 y luego agregando 5 dos veces, mientras que el número 15 no puede ser alcanzado de ninguna manera.

Aquí hay una solución recursiva:

```
function encontrarSolucion(objetivo) {
```

```
  function encontrar(actual, historia) {
    if (actual == objetivo) {
      return historia;
    } else if (actual > objetivo) {
      return null;
    } else {
      return encontrar(actual + 5, `(${historia} + 5)`) ||
             encontrar(actual * 3, `(${historia} * 3)`);
    }
  }
  return encontrar(1, "1");
}

console.log(encontrarSolucion(24));
// → (((1 * 3) + 5) * 3)
```

Ten en cuenta que este programa no necesariamente encuentra la secuencia de operaciones *mas corta*. Este está satisfecho cuando encuentra cualquier secuencia que funcione.

Está bien si no ves cómo funciona el programa de inmediato. Vamos a trabajar a través de él, ya que es un gran ejercicio de pensamiento recursivo.

La función interna `encontrar` es la que hace uso de la recursión real. Esta toma dos argumentos, el número actual y un string que registra cómo se ha alcanzado este número. Si encuentra una solución, devuelve un string que muestra cómo llegar al objetivo. Si no puede encontrar una solución a partir de este número, retorna `null`.

Para hacer esto, la función realiza una de tres acciones. Si el número actual es el número objetivo, la historia actual es una forma de llegar a ese objetivo, por lo que es retornada. Si el número actual es mayor que el objetivo, no tiene sentido seguir explorando esta rama ya que tanto agregar como multiplicar solo hara que el número sea mas grande, por lo que retorna `null`. Y finalmente, si aún estamos por debajo del número objetivo, la función intenta ambos caminos posibles que comienzan desde el número actual llamandose a sí misma dos veces, una para agregar y otra para multiplicar. Si la primera llamada devuelve algo que no es `null`, esta es retornada. De lo contrario, se retorna la segunda llamada, independientemente de si produce un string o el valor `null`.

Para comprender mejor cómo esta función produce el efecto que estamos buscando, veamos todas las llamadas a `encontrar` que se hacen cuando buscamos una solución para el número 13.

```
encontrar(1, "1")
```

```
encontrar(6, "(1 + 5)")
  encontrar(11, "((1 + 5) + 5)")
    encontrar(16, "(((1 + 5) + 5) + 5)")
      muy grande
    encontrar(33, "(((1 + 5) + 5) * 3)")
      muy grande
  encontrar(18, "((1 + 5) * 3)")
    muy grande
encontrar(3, "(1 * 3)")
  encontrar(8, "((1 * 3) + 5)")
    encontrar(13, "(((1 * 3) + 5) + 5)")
      ¡encontrado!
```

La indentación indica la profundidad de la pila de llamadas. La primera vez que `encontrar` es llamada, comienza llamandose a sí misma para explorar la solución que comienza con `(1 + 5)`. Esa llamada hara uso de la recursión aún más para explorar *cada* solución continuada que produzca un número menor o igual a el número objetivo. Como no encuentra uno que llegue al objetivo, retorna `null` a la primera llamada. Ahí el operador `||` genera la llamada que explora `(1 * 3)` para que esta suceda. Esta búsqueda tiene más suerte—su primera llamada recursiva, a través de *otra* llamada recursiva, encuentra al número objetivo. Esa llamada más interna retorna un string, y cada uno de los operadores `||` en las llamadas intermedias pasa ese string a lo largo, en última instancia retornando la solución.

Funciones crecientes

Hay dos formas más o menos naturales para que las funciones sean introducidas en los programas.

La primera es que te encuentras escribiendo código muy similar múltiples veces. Preferiríamos no hacer eso. Tener más código significa más espacio para que los errores se oculten y más material que leer para las personas que intenten entender el programa. Entonces tomamos la funcionalidad repetida, buscamos un buen nombre para ella, y la ponemos en una función.

La segunda forma es que encuentres que necesitas alguna funcionalidad que aún no has escrito y parece que merece su propia función. Comenzarás por nombrar a la función y luego escribirás su cuerpo. Incluso podrías comenzar a escribir código que use la función antes de que definas a la función en sí misma.

Que tan difícil te sea encontrar un buen nombre para una función es una buena indicación de cuán claro es el concepto que está tratando de envolver.

Veamos un ejemplo.

Queremos escribir un programa que imprima dos números, los números de vacas y pollos en una granja, con las palabras `Vacas` y `Pollos` después de ellos, y ceros acolchados antes de ambos números para que siempre tengan tres dígitos de largo.

```
007 Vacas
011 Pollos
```

Esto pide una función de dos argumentos—el numero de vacas y el numero de pollos. Vamos a programar.

```
function imprimirInventarioGranja(vacas, pollos) {
  let stringVaca = String(vacas);
  while (stringVaca.length < 3) {
    stringVaca = "0" + stringVaca;
  }
  console.log(`${stringVaca} Vacas`);
  let stringPollos = String(pollos);
  while (stringPollos.length < 3) {
    stringPollos = "0" + stringPollos;
  }
  console.log(`${stringPollos} Pollos`);
}
imprimirInventarioGranja(7, 11);
```

Escribir `.length` después de una expresión de string nos dará la longitud de dicho string. Por lo tanto, los ciclos `while` seguiran sumando ceros delante del string de numeros hasta que este tenga al menos tres caracteres de longitud.

Misión cumplida! Pero justo cuando estamos por enviar el código a la agricultora (junto con una considerable factura), ella nos llama y nos dice que ella también comenzó a criar cerdos, y que si no podríamos extender el software para imprimir cerdos también?

Claro que podemos. Pero justo cuando estamos en el proceso de copiar y pegar esas cuatro líneas una vez más, nos detenemos y reconsideramos. Tiene que haber una mejor manera. Aquí hay un primer intento:

```
function imprimirEtiquetaAlcochadaConCeros(numero, etiqueta) {
  let stringNumero = String(numero);
  while (stringNumero.length < 3) {
    stringNumero = "0" + stringNumero;
  }
```

```
  console.log(`${stringNumero} ${etiqueta}`);
}

function imprimirInventarioGranja(vacas, pollos, cerdos) {
  imprimirEtiquetaAlcochadaConCeros(vacas, "Vacas");
  imprimirEtiquetaAlcochadaConCeros(pollos, "Pollos");
  imprimirEtiquetaAlcochadaConCeros(cerdos, "Cerdos");
}

imprimirInventarioGranja(7, 11, 3);
```

Funciona! Pero ese nombre, `imprimirEtiquetaAlcochadaConCeros`, es un poco incómodo. Combina tres cosas—impresión, alcochar con ceros y añadir una etiqueta—en una sola función.

En lugar de sacar la parte repetida de nuestro programa al por mayor, intentemos elegir un solo *concepto*.

```
function alcocharConCeros(numero, amplitud) {
  let string = String(numero);
  while (string.length < amplitud) {
    string = "0" + string;
  }
  return string;
}

function imprimirInventarioGranja(vacas, pollos, cerdos) {
  console.log(`${alcocharConCeros(vacas, 3)} Vacas`);
  console.log(`${alcocharConCeros(pollos, 3)} Pollos`);
  console.log(`${alcocharConCeros(cerdos, 3)} Cerdos`);
}

imprimirInventarioGranja(7, 16, 3);
```

Una función con un nombre agradable y obvio como `alcocharConCeros` hace que sea más fácil de entender lo que hace para alguien que lee el código. Y tal función es útil en situaciones más alla de este programa en específico. Por ejemplo, podrías usarla para ayudar a imprimir tablas de números en una manera alineada.

Que tan inteligente y versátil *deberia* de ser nuestra función? Podríamos escribir cualquier cosa, desde una función terriblemente simple que solo pueda alcochar un número para que tenga tres caracteres de ancho, a un complicado sistema generalizado de formateo de números que maneje números fraccionar-

ios, números negativos, alineación de puntos decimales, relleno con diferentes caracteres, y así sucesivamente.

Un principio útil es no agregar mucho ingenio a menos que estes absolutamente seguro de que lo vas a necesitar. Puede ser tentador escribir "frameworks" generalizados para cada funcionalidad que encuentres. Resiste ese impulso. No realizarás ningún trabajo real de esta manera—solo estarás escribiendo código que nunca usarás.

Funciones y efectos secundarios

Las funciones se pueden dividir aproximadamente en aquellas que se llaman por su efectos secundarios y aquellas que son llamadas por su valor de retorno. (Aunque definitivamente también es posible tener tanto efectos secundarios como devolver un valor en una misma función.)

La primera función auxiliar en el ejemplo de la granja, `imprimirEtiquetaAlcochadaConCeros`, se llama por su efecto secundario: imprime una línea. La segunda versión, `alcocharConCeros`, se llama por su valor de retorno. No es coincidencia que la segunda sea útil en más situaciones que la primera. Las funciones que crean valores son más fáciles de combinar en nuevas formas que las funciones que directamente realizan efectos secundarios.

Una función *pura* es un tipo específico de función de producción-de-valores que no solo no tiene efectos secundarios pero que tampoco depende de los efectos secundarios de otro código—por ejemplo, no lee vinculaciones globales cuyos valores puedan cambiar. Una función pura tiene la propiedad agradable de que cuando se le llama con los mismos argumentos, siempre produce el mismo valor (y no hace nada más). Una llamada a tal función puede ser sustituida por su valor de retorno sin cambiar el significado del código. Cuando no estás seguro de que una función pura esté funcionando correctamente, puedes probarla simplemente llamándola, y saber que si funciona en ese contexto, funcionará en cualquier contexto. Las funciones no puras tienden a requerir más configuración para poder ser probadas.

Aún así, no hay necesidad de sentirse mal cuando escribas funciones que no son puras o de hacer una guerra santa para purgarlas de tu código. Los efectos secundarios a menudo son útiles. No habría forma de escribir una versión pura de `console.log`, por ejemplo, y `console.log` es bueno de tener. Algunas operaciones también son más fáciles de expresar de una manera eficiente cuando usamos efectos secundarios, por lo que la velocidad de computación puede ser una razón para evitar la pureza.

Resumen

Este capítulo te enseñó a escribir tus propias funciones. La palabra clave `function`, cuando se usa como una expresión, puede crear un valor de función. Cuando se usa como una declaración, se puede usar para declarar una vinculación y darle una función como su valor. Las funciones de flecha son otra forma más de crear funciones.

```
// Define f para sostener un valor de función
const f = function(a) {
  console.log(a + 2);
};

// Declara g para ser una función
function g(a, b) {
  return a * b * 3.5;
}

// Un valor de función menos verboso
let h = a => a % 3;
```

Un aspecto clave en para comprender a las funciones es comprender los alcances. Cada bloque crea un nuevo alcance. Los parámetros y vinculaciones declaradas en un determinado alcance son locales y no son visibles desde el exterior. Vinculaciones declaradas con `var` se comportan de manera diferente—terminan en el alcance de la función más cercana o en el alcance global.

Separar las tareas que realiza tu programa en diferentes funciones es util. No tendrás que repetirte tanto, y las funciones pueden ayudar a organizar un programa agrupando el código en piezas que hagan cosas especificas.

Ejercicios

Mínimo

El capítulo anterior introdujo la función estándar `Math.min` que devuelve su argumento más pequeño. Nosotros podemos construir algo como eso ahora. Escribe una función `min` que tome dos argumentos y retorne su mínimo.

Recursión

Hemos visto que `%` (el operador de residuo) se puede usar para probar si un número es par o impar usando `% 2` para ver si es divisible entre dos. Aquí hay

otra manera de definir si un número entero positivo es par o impar:

- Zero es par.
- Uno es impar.
- Para cualquier otro número N, su paridad es la misma que N - 2.

Define una función recursiva `esPar` que corresponda a esta descripción. La función debe aceptar un solo parámetro (un número entero, positivo) y devolver un Booleano.

Pruébalo con 50 y 75. Observa cómo se comporta con -1. Por qué? Puedes pensar en una forma de arreglar esto?

Conteo de frijoles

Puedes obtener el N-ésimo carácter, o letra, de un string escribiendo `"string"[N]`. El valor devuelto será un string que contiene solo un carácter (por ejemplo, `"f"`). El primer carácter tiene posición cero, lo que hace que el último se encuentre en la posición `string.length - 1`. En otras palabras, un string de dos caracteres tiene una longitud de 2, y sus carácteres tendrán las posiciones 0 y 1.

Escribe una función `contarFs` que tome un string como su único argumento y devuelva un número que indica cuántos caracteres “F” en mayúsculas haya en el string.

Despues, escribe una función llamada `contarCaracteres` que se comporte como `contarFs`, excepto que toma un segundo argumento que indica el carácter que debe ser contado (en lugar de contar solo caracteres “F” en mayúscula). Reescribe `contarFs` para que haga uso de esta nueva función.

"En dos ocasiones me han preguntado, 'Dinos, Sr. Babbage, si pones montos equivocadas en la máquina, saldrán las respuestas correctas? [...] No soy capaz de comprender correctamente el tipo de confusión de ideas que podrían provocar tal pregunta."

—Charles Babbage, Passages from the Life of a Philosopher (1864)

Chapter 4

Estructuras de Datos: Objetos y Arrays

Los números, los booleanos y los strings son los átomos que constituyen las estructuras de datos. Sin embargo, muchos tipos de información requieren más de un átomo. Los *objetos* nos permiten agrupar valores—incluidos otros objetos— para construir estructuras más complejas.

Los programas que hemos construido hasta ahora han estado limitados por el hecho de que estaban operando solo en tipos de datos simples. Este capítulo introducira estructuras de datos básicas. Al final de el, sabrás lo suficiente como para comenzar a escribir programas útiles.

El capítulo trabajara a través de un ejemplo de programación más o menos realista, presentando nuevos conceptos según se apliquen al problema en cuestión. El código de ejemplo a menudo se basara en funciones y vinculaciones que fueron introducidas anteriormente en el texto.

La caja de arena en línea para el libro (*eloquentjavascript.net/code*] proporciona una forma de ejecutar código en el contexto de un capítulo en específico. Si decides trabajar con los ejemplos en otro entorno, asegúrate de primero descargar el código completo de este capítulo desde la página de la caja de arena.

El Hombre Ardilla

De vez en cuando, generalmente entre las ocho y las diez de la noche, Jacques se encuentra a si mismo transformándose en un pequeño roedor peludo con una cola espesa.

Por un lado, Jacques está muy contento de no tener la licantropía clásica. Convertirse en una ardilla causa menos problemas que convertirse en un lobo. En lugar de tener que preocuparse por accidentalmente comerse al vecino (*eso* sería incómodo), le preocupa ser comido por el gato del vecino. Después de dos ocasiones en las que se despertó en una rama precariamente delgada de la copa de un roble, desnudo y desorientado, Jacques se ha dedicado a bloquear las puertas y ventanas de su habitación por la noche y pone algunas nueces en

el piso para mantenerse ocupado.

Eso se ocupa de los problemas del gato y el árbol. Pero Jacques preferiría deshacerse de su condición por completo. Las ocurrencias irregulares de la transformación lo hacen sospechar que estas podrían ser provocadas por algo en especifico. Por un tiempo, creyó que solo sucedia en los días en los que el había estado cerca de árboles de roble. Pero evitar los robles no detuvo el problema.

Cambiando a un enfoque más científico, Jacques ha comenzado a mantener un registro diario de todo lo que hace en un día determinado y si su forma cambio. Con esta información el espera reducir las condiciones que desencadenan las transformaciones.

Lo primero que el necesita es una estructura de datos para almacenar esta información.

Conjuntos de datos

Para trabajar con una porción de datos digitales, primero debemos encontrar una manera de representarlo en la memoria de nuestra máquina. Digamos, por ejemplo, que queremos representar una colección de los números 2, 3, 5, 7 y 11.

Podríamos ponernos creativos con los strings—después de todo, los strings pueden tener cualquier longitud, por lo que podemos poner una gran cantidad de datos en ellos—y usar `"2 3 5 7 11"` como nuestra representación. Pero esto es incómodo. Tendrías que extraer los dígitos de alguna manera y convertirlos a números para acceder a ellos.

Afortunadamente, JavaScript proporciona un tipo de datos específicamente para almacenar secuencias de valores. Es llamado *array* y está escrito como una lista de valores entre corchetes, separados por comas.

```
let listaDeNumeros = [2, 3, 5, 7, 11];
console.log(listaDeNumeros[2]);
// → 5
console.log(listaDeNumeros[0]);
// → 2
console.log(listaDeNumeros[2 - 1]);
// → 3
```

La notación para llegar a los elementos dentro de un array también utiliza corchetes. Un par de corchetes inmediatamente después de una expresión, con otra expresión dentro de ellos, buscará al elemento en la expresión de la izquierda que corresponde al *índice* dado por la expresión entre corchetes.

El primer índice de un array es cero, no uno. Entonces el primer elemento es alcanzado con `listaDeNumeros[0]`. El conteo basado en cero tiene una larga tradición en el mundo de la tecnología, y en ciertas maneras tiene mucho sentido, pero toma algo de tiempo acostumbrarse. Piensa en el índice como la cantidad de elementos a saltar, contando desde el comienzo del array.

PROPIEDADES

Hasta ahora hemos visto algunas expresiones sospechosas como `miString.length` (para obtener la longitud de un string) y `Math.max` (la función máxima) en capítulos anteriores. Estas son expresiones que acceden a la *propiedad* de algún valor. En el primer caso, accedemos a la propiedad `length` de el valor en `miString`. En el segundo, accedemos a la propiedad llamada `max` en el objeto `Math` (que es una colección de constantes y funciones relacionadas con las matemáticas).

Casi todos los valores de JavaScript tienen propiedades. Las excepciones son `null` y `undefined`. Si intentas acceder a una propiedad en alguno de estos no-valores, obtienes un error.

```
null.length;
// → TypeError: null has no properties
```

Las dos formas principales de acceder a las propiedades en JavaScript son con un punto y con corchetes. Tanto `valor.x` como `valor[x]` acceden una propiedad en `valor`—pero no necesariamente la misma propiedad. La diferencia está en cómo se interpreta `x`. Cuando se usa un punto, la palabra después del punto es el nombre literal de la propiedad. Cuando usas corchetes, la expresión entre corchetes es *evaluada* para obtener el nombre de la propiedad. Mientras `valor.x` obtiene la propiedad de `valor` llamada "x", `valor[x]` intenta evaluar la expresión `x` y usa el resultado, convertido en un string, como el nombre de la propiedad.

Entonces, si sabes que la propiedad que te interesa se llama *color*, dices `valor.color`. Si quieres extraer la propiedad nombrado por el valor mantenido en la vinculación `i`, dices `valor[i]`. Los nombres de las propiedades son strings. Pueden ser cualquier string, pero la notación de puntos solo funciona con nombres que se vean como nombres de vinculaciones válidos. Entonces, si quieres acceder a una propiedad llamada *2* o *Juan Perez*, debes usar corchetes: `valor[2]` o `valor["Juan Perez"]`.

Los elementos en un array son almacenados como propiedades del array,

usando números como nombres de propiedad. Ya que no puedes usar la notación de puntos con números, y que generalmente quieres utilizar una vinculación que contenga el índice de cualquier manera, debes de usar la notación de corchetes para llegar a ellos.

La propiedad `length` de un array nos dice cuántos elementos este tiene. Este nombre de propiedad es un nombre de vinculación válido, y sabemos su nombre en avance, así que para encontrar la longitud de un array, normalmente escribes `array.length` ya que es más fácil de escribir que `array["length"]`.

MÉTODOS

Ambos objetos de string y array contienen, además de la propiedad `length`, una serie de propiedades que tienen valores de función.

```
let ouch = "Ouch";
console.log(typeof ouch.toUpperCase);
// → function
console.log(ouch.toUpperCase());
// → OUCH
```

Cada string tiene una propiedad `toUpperCase` ("a mayúsculas"). Cuando se llame, regresará una copia del string en la que todas las letras han sido convertido a mayúsculas. También hay `toLowerCase` ("a minúsculas"), que hace lo contrario.

Curiosamente, a pesar de que la llamada a `toUpperCase` no pasa ningún argumento, la función de alguna manera tiene acceso al string `"Ouch"`, el valor de cuya propiedad llamamos. Cómo funciona esto se describe en el Capítulo 6.

Las propiedades que contienen funciones generalmente son llamadas *metodos* del valor al que pertenecen. Como en, "`toUpperCase` es un método de string".

Este ejemplo demuestra dos métodos que puedes usar para manipular arrays:

```
let secuencia = [1, 2, 3];
secuencia.push(4);
secuencia.push(5);
console.log(secuencia);
// → [1, 2, 3, 4, 5]
console.log(secuencia.pop());
// → 5
console.log(secuencia);
// → [1, 2, 3, 4]
```

El método `push` agrega valores al final de un array, y el el método `pop` hace lo contrario, eliminando el último valor en el array y retornandolo.

Estos nombres algo tontos son los términos tradicionales para las operaciones en una *pila*. Una pila, en programación, es una estructura de datos que te permite agregar valores a ella y volverlos a sacar en el orden opuesto, de modo que lo que se agregó de último se elimine primero. Estas son comunes en la programación—es posible que recuerdes la pila de llamadas en el capítulo anterior, que es una instancia de la misma idea.

Objetos

De vuelta al Hombre-Ardilla. Un conjunto de entradas diarias puede ser representado como un array. Pero estas entradas no consisten en solo un número o un string—cada entrada necesita almacenar una lista de actividades y un valor booleano que indica si Jacques se convirtió en una ardilla o no. Idealmente, nos gustaría agrupar estos en un solo valor y luego agrupar estos valores en un array de registro de entradas.

Los valores del tipo *objeto* son colecciones arbitrarias de propiedades. Una forma de crear un objeto es mediante el uso de llaves como una expresión.

```
let dia1 = {
  ardilla: false,
  eventos: ["trabajo", "toque un arbol", "pizza", "salir a correr"]
};
console.log(dia1.ardilla);
// → false
console.log(dia1.lobo);
// → undefined
dia1.lobo = false;
console.log(dia1.lobo);
// → false
```

Dentro de las llaves, hay una lista de propiedades separadas por comas. Cada propiedad tiene un nombre seguido de dos puntos y un valor. Cuando un objeto está escrito en varias líneas, indentar como en el ejemplo ayuda con la legibilidad. Las propiedades cuyos nombres no sean nombres válidos de vinculaciones o números válidos deben estar entre comillas.

```
let descripciones = {
  trabajo: "Fui a trabajar",
  "toque un arbol": "Toque un arbol"
```

```
};
```

Esto significa que las llaves tienen *dos* significados en JavaScript. Al comienzo de una declaración, comienzan un bloque de declaraciones. En cualquier otra posición, describen un objeto. Afortunadamente, es raramente útil comenzar una declaración con un objeto en llaves, por lo que la ambigüedad entre estas dos acciones no es un gran problema.

Leer una propiedad que no existe te dará el valor `undefined`.

Es posible asignarle un valor a una expresión de propiedad con un operador `=`. Esto reemplazará el valor de la propiedad si ya tenia uno o crea una nueva propiedad en el objeto si no fuera así.

Para volver brevemente a nuestro modelo de vinculaciones como tentáculos—Las vinculaciones de propiedad son similares. Ellas *agarran* valores, pero otras vinculaciones y propiedades pueden estar agarrando esos mismos valores. Puedes pensar en los objetos como pulpos con cualquier cantidad de tentáculos, cada uno de los cuales tiene un nombre tatuado en él.

El operador `delete` ("eliminar") corta un tentáculo de dicho pulpo. Es un operador unario que, cuando se aplica a la propiedad de un objeto, eliminará la propiedad nombrada de dicho objeto. Esto no es algo que hagas todo el tiempo, pero es posible.

```
let unObjeto = {izquierda: 1, derecha: 2};
console.log(unObjeto.izquierda);
// → 1
delete unObjeto.izquierda;
console.log(unObjeto.izquierda);
// → undefined
console.log("izquierda" in unObjeto);
// → false
console.log("derecha" in unObjeto);
// → true
```

El operador binario `in` ("en"), cuando se aplica a un string y un objeto, te dice si ese objeto tiene una propiedad con ese nombre. La diferencia entre darle un valor de `undefined` a una propiedad y eliminarla realmente es que, en el primer caso, el objeto todavía *tiene* la propiedad (solo que no tiene un valor muy interesante), mientras que en el segundo caso la propiedad ya no está presente e `in` retornara `false`.

Para saber qué propiedades tiene un objeto, puedes usar la función `Object.keys`. Le das un objeto y devuelve un array de strings—los nombres de las

propiedades del objeto.

```
console.log(Object.keys({x: 0, y: 0, z: 2}));
// → ["x", "y", "z"]
```

Hay una función `Object.assign` que copia todas las propiedades de un objeto a otro.

```
let objetoA = {a: 1, b: 2};
Object.assign(objetoA, {b: 3, c: 4});
console.log(objetoA);
// → {a: 1, b: 3, c: 4}
```

Los arrays son, entonces, solo un tipo de objeto especializado para almacenar secuencias de cosas. Si evalúas `typeof []`, este produce `"object"`. Podrias imaginarlos como pulpos largos y planos con todos sus tentáculos en una fila ordenada, etiquetados con números.

Representaremos el diario de Jacques como un array de objetos.

```
let diario = [
  {eventos: ["trabajo", "toque un arbol", "pizza",
             "sali a correr", "television"],
   ardilla: false},
  {eventos: ["trabajo", "helado", "coliflor",
             "lasaña", "toque un arbol", "me cepille los dientes"],
   ardilla: false},
  {eventos: ["fin de semana", "monte la bicicleta", "descanso", "
     nueces",
             "cerveza"],
   ardilla: true},
  /* y asi sucesivamente... */
];
```

MUTABILIDAD

Llegaremos a la programación real *pronto*. Pero primero, hay una pieza más de teoría por entender.

Vimos que los valores de objeto pueden ser modificados. Los tipos de valores discutidos en capítulos anteriores, como números, strings y booleanos, son todos *inmutables*—es imposible cambiar los valores de aquellos tipos. Puedes

combinarlos y obtener nuevos valores a partir de ellos, pero cuando tomas un valor de string específico, ese valor siempre será el mismo. El texto dentro de él no puede ser cambiado. Si tienes un string que contiene `"gato"`, no es posible que otro código cambie un carácter en tu string para que deletree `"rato"`.

Los objetos funcionan de una manera diferente. Tu *puedes* cambiar sus propiedades, haciendo que un único valor de objeto tenga contenido diferente en diferentes momentos.

Cuando tenemos dos números, 120 y 120, podemos considerarlos el mismo número precisamente, ya sea que hagan referencia o no a los mismos bits físicos. Con los objetos, hay una diferencia entre tener dos referencias a el mismo objeto y tener dos objetos diferentes que contengan las mismas propiedades. Considera el siguiente código:

```
let objeto1 = {valor: 10};
let objeto2 = objeto1;
let objeto3 = {valor: 10};

console.log(objeto1 == objeto2);
// → true
console.log(objeto1 == objeto3);
// → false

objeto1.valor = 15;
console.log(objeto2.valor);
// → 15
console.log(objeto3.valor);
// → 10
```

Las vinculaciones `objeto1` y `objeto2` agarran el *mismo* objeto, que es la razon por la cual cambiar `objeto1` también cambia el valor de `objeto2`. Se dice que tienen la misma *identidad*. La vinculación `objeto3` apunta a un objeto diferente, que inicialmente contiene las mismas propiedades que `objeto1` pero vive una vida separada.

Las vinculaciones también pueden ser cambiables o constantes, pero esto es independiente de la forma en la que se comportan sus valores. Aunque los valores numéricos no cambian, puedes usar una vinculación `let` para hacer un seguimiento de un número que cambia al cambiar el valor al que apunta la vinculación. Del mismo modo, aunque una vinculación `const` a un objeto no pueda ser cambiada en si misma y continuará apuntando al mismo objeto, los *contenidos* de ese objeto pueden cambiar.

```
const puntuacion = {visitantes: 0, locales: 0};
```

```
// Esto esta bien
puntuacion.visitantes = 1;
// Esto no esta permitido
puntuacion = {visitantes: 1, locales: 1};
```

Cuando comparas objetos con el operador `==` en JavaScript, este los compara por identidad: producirá `true` solo si ambos objetos son precisamente el mismo valor. Comparar diferentes objetos retornara `false`, incluso si tienen propiedades idénticas. No hay una operación de comparación "profunda" incorporada en JavaScript, que compare objetos por contenidos, pero es posible que la escribas tu mismo (que es uno de los ejercicios al final de este capítulo).

El diario del licántropo

Asi que Jacques inicia su intérprete de JavaScript y establece el entorno que necesita para mantener su diario.

```
let diario = [];

function añadirEntrada(eventos, ardilla) {
  diario.push({eventos, ardilla});
}
```

Ten en cuenta que el objeto agregado al diario se ve un poco extraño. En lugar de declarar propiedades como `eventos: eventos`, simplemente da un nombre de propiedad. Este es un atajo que representa lo mismo—si el nombre de propiedad en la notación de llaves no es seguido por un valor, su el valor se toma de la vinculación con el mismo nombre.

Entonces, todas las noches a las diez—o algunas veces a la mañana siguiente, después de bajar del estante superior de su biblioteca—Jacques registra el día.

So then, every evening at ten—or sometimes the next morning, after climbing down from the top shelf of his bookcase—Jacques records the day.

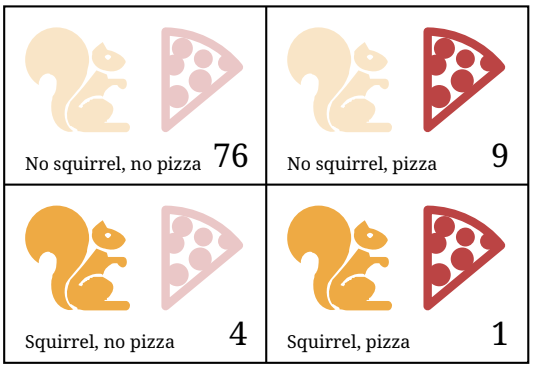
```
añadirEntrada(["trabajo", "toque un arbol", "pizza", "sali a correr
   ",
          "television"], false);
añadirEntrada(["trabajo", "helado", "coliflor", "lasaña",
          "toque un arbol", "me cepille los dientes"], false);
añadirEntrada(["fin de semana", "monte la bicicleta", "descanso", "
   nueces",
          "cerveza"], true);
```

Una vez que tiene suficientes puntos de datos, tiene la intención de utilizar estadísticas para encontrar cuál de estos eventos puede estar relacionado con la transformación a ardilla.

La *correlación* es una medida de dependencia entre variables estadísticas. Una variable estadística no es lo mismo que una variable de programación. En las estadísticas, normalmente tienes un conjunto de *medidas*, y cada variable se mide para cada medida. La correlación entre variables generalmente se expresa como un valor que varia de -1 a 1. Una correlación de cero significa que las variables no estan relacionadas. Una correlación de uno indica que las dos están perfectamente relacionadas—si conoces una, también conoces la otra. Uno negativo también significa que las variables están perfectamente relacionadas pero que son opuestas—cuando una es verdadera, la otra es falsa.

Para calcular la medida de correlación entre dos variables booleanas, podemos usar el *coeficiente phi* (φ). Esta es una fórmula cuya entrada es una tabla de frecuencias que contiene la cantidad de veces que las diferentes combinaciones de las variables fueron observadas. El resultado de la fórmula es un número entre -1 y 1 que describe la correlación.

Podríamos tomar el evento de comer pizza y poner eso en una tabla de frecuencias como esta, donde cada número indica la cantidad de veces que ocurrió esa combinación en nuestras mediciones:



Si llamamos a esa tabla n, podemos calcular φ usando la siguiente fórmula:

$$\varphi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}} \tag{4.1}$$

(Si en este momento estas bajando el libro para enfocarte en un terrible flashback a la clase de matemática de 10° grado—espera! No tengo la intención de torturarte con infinitas páginas de notación críptica—solo esta fórmula para ahora. E incluso con esta, todo lo que haremos es convertirla en JavaScript.)

La notación n_{01} indica el número de mediciones donde la primera variable

(ardilla) es falso (0) y la segunda variable (pizza) es verdadera (1). En la tabla de pizza, n_{01} es 9.

El valor $n_{1\bullet}$ se refiere a la suma de todas las medidas donde la primera variable es verdadera, que es 5 en la tabla de ejemplo. Del mismo modo, $n_{\bullet 0}$ se refiere a la suma de las mediciones donde la segunda variable es falsa.

Entonces para la tabla de pizza, la parte arriba de la línea de división (el dividendo) sería $1\times76-4\times9 = 40$, y la parte inferior (el divisor) sería la raíz cuadrada de $5\times85\times10\times80$, o $\sqrt{340000}$. Esto da $\varphi \approx 0.069$, que es muy pequeño. Comer pizza no parece tener influencia en las transformaciones.

Calculando correlación

Podemos representar una tabla de dos-por-dos en JavaScript con un array de cuatro elementos (`[76, 9, 4, 1]`). También podríamos usar otras representaciones, como un array que contiene dos arrays de dos elementos (`[[76, 9], [4, 1]]`) o un objeto con nombres de propiedad como `"11"` y `"01"`, pero el array plano es simple y hace que las expresiones que acceden a la tabla agradablemente cortas. Interpretaremos los índices del array como número binarios de dos-bits , donde el dígito más a la izquierda (más significativo) se refiere a la variable ardilla y el digito mas a la derecha (menos significativo) se refiere a la variable de evento. Por ejemplo, el número binario `10` se refiere al caso en que Jacques se convirtió en una ardilla, pero el evento (por ejemplo, "pizza") no ocurrió. Esto ocurrió cuatro veces. Y dado que el `10` binario es 2 en notación decimal, almacenaremos este número en el índice 2 del array.

Esta es la función que calcula el coeficiente φ de tal array:

```
function phi(tabla) {
  return (tabla[3] * tabla[0] - tabla[2] * tabla[1]) /
    Math.sqrt((tabla[2] + tabla[3]) *
              (tabla[0] + tabla[1]) *
              (tabla[1] + tabla[3]) *
              (tabla[0] + tabla[2]));
}

console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

Esta es una traducción directa de la fórmula φ a JavaScript. `Math.sqrt` es la función de raíz cuadrada, proporcionada por el objeto `Math` en un entorno de JavaScript estándar. Tenemos que sumar dos campos de la tabla para

obtener campos como $n_{1\bullet}$ porque las sumas de filas o columnas no se almacenan directamente en nuestra estructura de datos.

Jacques mantuvo su diario por tres meses. El conjunto de datos resultante está disponible en la caja de arena para este capítulo(*eloquentjavascript.net/code#4*), donde se almacena en la vinculación `JOURNAL`, y en un archivo descargable.

Para extraer una tabla de dos por dos para un evento en específico del diario, debemos hacer un ciclo a traves de todas las entradas y contar cuántas veces ocurre el evento en relación a las transformaciones de ardilla.

```
function tablaPara(evento, diario) {
  let tabla = [0, 0, 0, 0];
  for (let i = 0; i < diario.length; i++) {
    let entrada = diario[i], index = 0;
    if (entrada.eventos.includes(evento)) index += 1;
    if (entrada.ardilla) index += 2;
    tabla[index] += 1;
  }
  return tabla;
}

console.log(tablaPara("pizza", JOURNAL));
// → [76, 9, 4, 1]
```

Los array tienen un método `includes` ("incluye") que verifica si un valor dado existe en el array. La función usa eso para determinar si el nombre del evento en el que estamos interesados forma parte de la lista de eventos para un determinado día.

El cuerpo del ciclo en `tablaPara` determina en cual caja de la tabla cae cada entrada del diario al verificar si la entrada contiene el evento específico que nos interesa y si el evento ocurre junto con un incidente de ardilla. El ciclo luego agrega uno a la caja correcta en la tabla.

Ahora tenemos las herramientas que necesitamos para calcular las correlaciónes individuales. El único paso que queda es encontrar una correlación para cada tipo de evento que se escribio en el diario y ver si algo se destaca.

Ciclos de array

En la función `tablaPara`, hay un ciclo como este:

```
for (let i = 0; i < DIARIO.length; i++) {
```

```
  let entrada = DIARIO[i];
  // Hacer con algo con la entrada
}
```

Este tipo de ciclo es común en JavaScript clasico—ir a traves de los arrays un elemento a la vez es algo que surge mucho, y para hacer eso correrias un contador sobre la longitud del array y elegirías cada elemento en turnos.

Hay una forma más simple de escribir tales ciclos en JavaScript moderno.

```
for (let entrada of DIARIO) {
  console.log(`${entrada.eventos.length} eventos.`);
}
```

Cuando un ciclo `for` se vea de esta manera, con la palabra `of` ("de") después de una definición de variable, recorrerá los elementos del valor dado después `of`. Esto funciona no solo para arrays, sino también para strings y algunas otras estructuras de datos. Vamos a discutir *como* funciona en el Capítulo 6.

El análisis final

Necesitamos calcular una correlación para cada tipo de evento que ocurra en el conjunto de datos. Para hacer eso, primero tenemos que *encontrar* cada tipo de evento.

```
function eventosDiario(diario) {
  let eventos = [];
  for (let entrada of diario) {
    for (let evento of entrada.eventos) {
      if (!eventos.includes(evento)) {
        eventos.push(evento);
      }
    }
  }
  return eventos;
}

console.log(eventosDiario(DIARIO));
// → ["zanahoria", "ejercicio", "fin de semana", "pan", …]
```

Yendo a traves de todos los eventos, y agregando aquellos que aún no están en allí en el array `eventos`, la función recolecta cada tipo de evento.

Usando eso, podemos ver todos las correlaciones.

```
for (let evento of eventosDiario(DIARIO)) {
  console.log(evento + ":", phi(tablaPara(evento, DIARIO)));
}
// → zanahoria:      0.0140970969
// → ejercicio:      0.0685994341
// → fin de semana:  0.1371988681
// → pan:           -0.0757554019
// → pudin:         -0.0648203724
// and so on...
```

La mayoría de las correlaciones parecen estar cercanas a cero. Come zanahorias, pan o pudín aparentemente no desencadena la licantropía de ardilla. *Parece* ocurrir un poco más a menudo los fines de semana. Filtremos los resultados para solo mostrar correlaciones mayores que 0.1 o menores que -0.1.

```
for (let evento of eventosDiario(DIARIO)) {
  let correlacion = phi(tablaPara(evento, DIARIO));
  if (correlacion > 0.1 || correlacion < -0.1) {
    console.log(evento + ":", correlacion);
  }
}
// → fin de semana:           0.1371988681
// → me cepille los dientes: -0.3805211953
// → dulces:                  0.1296407447
// → trabajo:                -0.1371988681
// → spaghetti:               0.2425356250
// → leer:                    0.1106828054
// → nueces:                  0.5902679812
```

A-ha! Hay dos factores con una correlación que es claramente más fuerte que las otras. Comer nueces tiene un fuerte efecto positivo en la posibilidad de convertirse en una ardilla, mientras que cepillarse los dientes tiene un significativo efecto negativo.

Interesante. Intentemos algo.

```
for (let entrada of DIARIO) {
  if (entrada.eventos.includes("nueces") &&
     !entrada.eventos.includes("me cepille los dientes")) {
    entrada.eventos.push("dientes con nueces");
  }
}
```

```
console.log(phi(tablaPara("dientes con nueces", DIARIO)));
// → 1
```

Ese es un resultado fuerte. El fenómeno ocurre precisamente cuando Jacques come nueces y no se cepilla los dientes. Si tan solo él no hubiese sido tan flojo con su higiene dental, él nunca habría notado su aflicción.

Sabiendo esto, Jacques deja de comer nueces y descubre que sus transformaciones no vuelven.

Durante algunos años, las cosas van bien para Jacques. Pero en algún momento él pierde su trabajo. Porque vive en un país desagradable donde no tener trabajo significa que no tiene servicios médicos, se ve obligado a trabajar con a circo donde actua como *El Increible Hombre-Ardilla*, llenando su boca con mantequilla de maní antes de cada presentación.

Un día, harto de esta existencia lamentable, Jacques no puede cambiar de vuelta a su forma humana, salta a través de una grieta en la carpa del circo, y se desvanece en el bosque. Nunca se le ve de nuevo.

Arrayología avanzada

Antes de terminar el capítulo, quiero presentarte algunos conceptos extras relacionados a los objetos. Comenzaré introduciendo algunos en métodos de arrays útiles generalmente.

Vimos `push` y `pop`, que agregan y removen elementos en el final de un array, anteriormente en este capítulo. Los métodos correspondientes para agregar y remover cosas en el comienzo de un array se llaman `unshift` y `shift`.

```
let listaDeTareas = [];
function recordar(tarea) {
  listaDeTareas.push(tarea);
}
function obtenerTarea() {
  return listaDeTareas.shift();
}
function recordarUrgentemente(tarea) {
  listaDeTareas.unshift(tarea);
}
```

Ese programa administra una cola de tareas. Agregas tareas al final de la cola al llamar `recordar("verduras")`, y cuando estés listo para hacer algo, llamas a `obtenerTarea()` para obtener (y eliminar) el elemento frontal de la cola.

La función `recordarUrgentemente` también agrega una tarea pero la agrega al frente en lugar de a la parte posterior de la cola.

Para buscar un valor específico, los arrays proporcionan un método `indexOf` ("indice de"). Este busca a través del array desde el principio hasta el final y retorna el índice en el que se encontró el valor solicitado—o -1 si este no fue encontrado. Para buscar desde el final en lugar del inicio, hay un método similar llamado `lastIndexOf` ("ultimo indice de").

```
console.log([1, 2, 3, 2, 1].indexOf(2));
// → 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// → 3
```

Tanto `indexOf` como `lastIndexOf` toman un segundo argumento opcional que indica dónde comenzar la búsqueda.

Otro método fundamental de array es `slice` ("rebanar"), que toma índices de inicio y fin y retorna un array que solo tiene los elementos entre ellos. El índice de inicio es inclusivo, el índice final es exclusivo.

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// → [2, 3, 4]
```

Cuando no se proporcione el índice final, `slice` tomará todos los elementos después del índice de inicio. También puedes omitir el índice de inicio para copiar todo el array.

El método `concat` ("concatenar") se puede usar para unir arrays y asi crear un nuevo array, similar a lo que hace el operador `+` para los strings.

El siguiente ejemplo muestra tanto `concat` como `slice` en acción. Toma un array y un índice, y retorna un nuevo array que es una copia del array original pero eliminando al elemento en el índice dado:

```
function remover(array, indice) {
  return array.slice(0, indice)
    .concat(array.slice(indice + 1));
}
console.log(remover(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

Si a `concat` le pasas un argumento que no es un array, ese valor sera agregado

al nuevo array como si este fuera un array de un solo elemento.

Strings y sus propiedades

Podemos leer propiedades como `length` y `toUpperCase` de valores string. Pero si intentas agregar una nueva propiedad, esta no se mantiene.

```
let kim = "Kim";
kim.edad = 88;
console.log(kim.edad);
// → undefined
```

Los valores de tipo string, número, y Booleano no son objetos, y aunque el lenguaje no se queja si intentas establecer nuevas propiedades en ellos, en realidad no almacena esas propiedades. Como se mencionó antes, tales valores son inmutables y no pueden ser cambiados.

Pero estos tipos tienen propiedades integradas. Cada valor de string tiene un numero de metodos. Algunos muy útiles son `slice` e `indexOf`, que se parecen a los métodos de array de los mismos nombres.

```
console.log("panaderia".slice(0, 3));
// → pan
console.log("panaderia".indexOf("a"));
// → 1
```

Una diferencia es que el `indexOf` de un string puede buscar por un string que contenga más de un carácter, mientras que el método correspondiente al array solo busca por un elemento único.

```
console.log("uno dos tres".indexOf("tres"));
// → 8
```

El método `trim` ("recortar") elimina los espacios en blanco (espacios, saltos de linea, tabulaciones y caracteres similares) del inicio y final de un string.

```
console.log("  okey \n ".trim());
// → okey
```

La función `alcocharConCeros` del capítulo anterior también existe como un método. Se llama `padStart` ("alcohar inicio") y toma la longitud deseada y el

carácter de relleno como argumentos.

```
console.log(String(6).padStart(3, "0"));
// → 006
```

Puedes dividir un string en cada ocurrencia de otro string con el metodo `split` (“dividir”), y unirlo nuevamente con `join` (“unir”).

```
let oracion = "Los pajaros secretarios se especializan en pisotear";
let palabras = oracion.split(" ");
console.log(palabras);
// → ["Los", "pajaros", "secretarios", "se", "especializan", "en", "
   pisotear"]
console.log(palabras.join(". "));
// → Los. pajaros. secretarios. se. especializan. en. pisotear
```

Se puede repetir un string con el método `repeat` (“repetir”), el cual crea un nuevo string que contiene múltiples copias concatenadas del string original.

```
console.log("LA".repeat(3));
// → LALALA
```

Ya hemos visto la propiedad `length` en los valores de tipo string. Acceder a los caracteres individuales en un string es similar a acceder a los elementos de un array (con una diferencia que discutiremos en el Capítulo 6).

```
let string = "abc";
console.log(string.length);
// → 3
console.log(string[1]);
// → b
```

Parámetros restantes

Puede ser útil para una función aceptar cualquier cantidad de argumentos. Por ejemplo, `Math.max` calcula el máximo de *todos* los argumentos que le son dados.

Para escribir tal función, pones tres puntos antes del ultimo parámetro de la función, asi:

```
function maximo(...numeros) {
```

```
  let resultado = -Infinity;
  for (let numero of numeros) {
    if (numero > resultado) resultado = numero;
  }
  return resultado;
}
console.log(maximo(4, 1, 9, -2));
// → 9
```

Cuando se llame a una función como esa, el *parámetro restante* está vinculado a un array que contiene todos los argumentos adicionales. Si hay otros parámetros antes que él, sus valores no seran parte de ese array. Cuando, tal como en `maximo`, sea el único parámetro, contendrá todos los argumentos.

Puedes usar una notación de tres-puntos similar para *llamar* una función con un array de argumentos.

```
let numeros = [5, 1, 7];
console.log(max(...numeros));
// → 7
```

Esto "extiende" al array en la llamada de la función, pasando sus elementos como argumentos separados. Es posible incluir un array de esa manera, junto con otros argumentos, como en `max(9, ...numeros, 2)`.

La notación de corchetes para crear arrays permite al operador de tres-puntos extender otro array en el nuevo array:

Square bracket array notation similarly allows the triple-dot operator to spread another array into the new array:

```
let palabras = ["nunca", "entenderas"];
console.log(["tu", ...palabras, "completamente"]);
// → ["tu", "nunca", "entenderas", "completamente"]
```

El objeto Math

Como hemos visto, `Math` es una bolsa de sorpresas de utilidades relacionadas a los numeros, como `Math.max` (máximo), `Math.min` (mínimo) y `Math.sqrt` (raíz cuadrada).

El objeto `Math` es usado como un contenedor que agrupa un grupo de funcionalidades relacionadas. Solo hay un objeto `Math`, y casi nunca es útil como

un valor. Más bien, proporciona un *espacio de nombre* para que todos estas funciones y valores no tengan que ser vinculaciones globales.

Tener demasiadas vinculaciones globales "contamina" el espacio de nombres. Cuanto más nombres hayan sido tomados, es más probable que accidentalmente sobrescribas el valor de algunas vinculaciones existentes. Por ejemplo, no es es poco probable que quieras nombrar algo `max` en alguno de tus programas. Dado que la función `max` ya incorporada en JavaScript está escondida dentro del Objeto `Math`, no tenemos que preocuparnos por sobrescribirla.

Muchos lenguajes te detendrán, o al menos te advertirán, cuando estes por definir una vinculación con un nombre que ya este tomado. JavaScript hace esto para vinculaciones que hayas declarado con `let` o`const` pero-perversamente-no para vinculaciones estándar, ni para vinculaciones declaradas con `var` o `function`.

De vuelta al objeto `Math`. Si necesitas hacer trigonometría, `Math` te puede ayudar. Contiene `cos` (coseno), `sin` (seno) y `tan` (tangente), así como sus funciones inversas, `acos`, `asin`, y `atan`, respectivamente. El número π (pi)—o al menos la aproximación más cercano que cabe en un número de JavaScript—está disponible como `Math.PI`. Hay una vieja tradición en la programación de escribir los nombres de los valores constantes en mayúsculas.

```
function puntoAleatorioEnCirculo(radio) {
  let angulo = Math.random() * 2 * Math.PI;
  return {x: radio * Math.cos(angulo),
          y: radio * Math.sin(angulo)};
}
console.log(puntoAleatorioEnCirculo(2));
// → {x: 0.3667, y: 1.966}
```

Si los senos y los cosenos son algo con lo que no estas familiarizado, no te preocupes. Cuando se usen en este libro, en el Capítulo 14, te los explicaré.

El ejemplo anterior usó `Math.random`. Esta es una función que retorna un nuevo número pseudoaleatorio entre cero (inclusivo) y uno (exclusivo) cada vez que la llamas.

```
console.log(Math.random());
// → 0.36993729369714856
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

Aunque las computadoras son máquinas deterministas—siempre reaccionan de la misma manera manera dada la misma entrada—es posible hacer que produzcan números que parecen aleatorios. Para hacer eso, la máquina mantiene algun valor escondido, y cada vez que le pidas un nuevo número aleatorio, realiza calculos complicados en este valor oculto para crear un nuevo valor. Esta almacena un nuevo valor y retorna un número derivado de él. De esta manera, puede producir números nuevos y difíciles de predecir de una manera que *parece* aleatoria.

Si queremos un número entero al azar en lugar de uno fraccionario, podemos usar `Math.floor` (que redondea hacia abajo al número entero más cercano) con el resultado de `Math.random`.

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

Multiplicar el número aleatorio por 10 nos da un número mayor que o igual a cero e inferior a 10. Como `Math.floor` redondea hacia abajo, esta expresión producirá, con la misma probabilidad, cualquier número desde 0 hasta 9.

También están las funciones `Math.ceil` (que redondea hacia arriba hasta llegar al número entero mas cercano), `Math.round` (al número entero más cercano), y `Math.abs`, que toma el valor absoluto de un número, lo que significa que niega los valores negativos pero deja los positivos tal y como estan.

Desestructurar

Volvamos a la función `phi` por un momento:

```
function phi(tabla) {
  return (tabla[3] * tabla[0] - tabla[2] * tabla[1]) /
    Math.sqrt((tabla[2] + tabla[3]) *
              (tabla[0] + tabla[1]) *
              (tabla[1] + tabla[3]) *
              (tabla[0] + tabla[2]));
}
```

Una de las razones por las que esta función es incómoda de leer es que tenemos una vinculación apuntando a nuestro array, pero preferiríamos tener vinculaciones para los *elementos* del array, es decir, `let n00 = tabla[0]` y así sucesivamente. Afortunadamente, hay una forma concisa de hacer esto en JavaScript.

```
function phi([n00, n01, n10, n11]) {
  return (n11 * n00 - n10 * n01) /
    Math.sqrt((n10 + n11) * (n00 + n01) *
              (n01 + n11) * (n00 + n10));
}
```

Esto también funciona para vinculaciones creadas con `let`, `var`, o `const`. Si sabes que el valor que estas vinculando es un array, puedes usar corchetes para "mirar dentro" del valor, y asi vincular sus contenidos.

Un truco similar funciona para objetos, utilizando llaves en lugar de corchetes.

```
let {nombre} = {nombre: "Faraji", edad: 23};
console.log(nombre);
// → Faraji
```

Ten en cuenta que si intentas desestructurar `null` o `undefined`, obtendrás un error, igual como te pasaria si intentaras acceder directamente a una propiedad de esos valores.

JSON

Ya que las propiedades solo agarran su valor, en lugar de contenerlo, los objetos y arrays se almacenan en la memoria de la computadora como secuencias de bits que contienen las *direcciónes*—el lugar en la memoria—de sus contenidos. Asi que un array con otro array dentro de el consiste en (al menos) una región de memoria para el array interno, y otra para el array externo, que contiene (entre otras cosas) un número binario que representa la posición del array interno.

Si deseas guardar datos en un archivo para más tarde, o para enviarlo a otra computadora a través de la red, tienes que convertir de alguna manera estos enredos de direcciones de memoria a una descripción que se puede almacenar o enviar. Supongo, que *podrías* enviar toda la memoria de tu computadora junto con la dirección del valor que te interesa, pero ese no parece el mejor enfoque.

Lo que podemos hacer es *serializar* los datos. Eso significa que son convertidos a una descripción plana. Un formato de serialización popular llamado *JSON* (pronunciado "Jason"), que significa JavaScript Object Notation ("Notación de Objetos JavaScript"). Es ampliamente utilizado como un formato de almacenamiento y comunicación de datos en la Web, incluso en otros lenguajes diferentes a JavaScript.

JSON es similar a la forma en que JavaScript escribe arrays y objetos, con algunas restricciones. Todos los nombres de propiedad deben estar rodeados por

comillas dobles, y solo se permiten expresiones de datos simples—sin llamadas a función, vinculaciones o cualquier otra cosa que involucre computaciones reales. Los comentarios no están permitidos en JSON.

Una entrada de diario podria verse así cuando se representa como datos JSON:

```
{
  "ardilla": false,
  "eventos": ["trabajo", "toque un arbol", "pizza", "sali a correr"]
}
```

JavaScript nos da las funciones `JSON.stringify` y `JSON.parse` para convertir datos hacia y desde este formato. El primero toma un valor en JavaScript y retorna un string codificado en JSON. La segunda toma un string como ese y lo convierte al valor que este codifica.

```
let string = JSON.stringify({ardilla: false,
                             eventos: ["fin de semana"]});
console.log(string);
// → {"ardilla":false,"eventos":["fin de semana"]}
console.log(JSON.parse(string).eventos);
// → ["fin de semana"]
```

Resumen

Los objetos y arrays (que son un tipo específico de objeto) proporcionan formas de agrupar varios valores en un solo valor. Conceptualmente, esto nos permite poner un montón de cosas relacionadas en un bolso y correr alrededor con el bolso, en lugar de envolver nuestros brazos alrededor de todas las cosas individuales, tratando de aferrarnos a ellas por separado.

La mayoría de los valores en JavaScript tienen propiedades, las excepciones son `null` y `undefined`. Se accede a las propiedades usando `valor.propiedad` o `valor["propiedad"]`. Los objetos tienden a usar nombres para sus propiedades y almacenar más o menos un conjunto fijo de ellos. Los arrays, por el otro lado, generalmente contienen cantidades variables de valores conceptualmente idénticos y usa números (comenzando desde 0) como los nombres de sus propiedades.

Hay *algunas* propiedades con nombre en los arrays, como `length` y un numero de metodos. Los métodos son funciones que viven en propiedades y (por lo general) actuan sobre el valor del que son una propiedad.

Puedes iterar sobre los arrays utilizando un tipo especial de ciclo `for`—`for (let elemento of array)`.

Ejercicios

La suma de un rango

La introducción de este libro aludía a lo siguiente como una buena forma de calcular la suma de un rango de números:

```
console.log(suma(rango(1, 10)));
```

Escribe una función `rango` que tome dos argumentos, `inicio` y `final`, y retorne un array que contenga todos los números desde `inicio` hasta (e incluyendo) `final`.

Luego, escribe una función `suma` que tome un array de números y retorne la suma de estos números. Ejecuta el programa de ejemplo y ve si realmente retorna 55.

Como una misión extra, modifica tu función `rango` para tomar un tercer argumento opcional que indique el valor de "paso" utilizado para cuando construyas el array. Si no se da ningún paso, los elementos suben en incrementos de uno, correspondiedo al comportamiento anterior. La llamada a la función `rango(1, 10, 2)` deberia retornar `[1, 3, 5, 7, 9]`. Asegúrate de que también funcione con valores de pasos negativos para que `rango(5, 2, -1)` produzca `[5, 4, 3, 2]`.

Revirtiendo un array

Los arrays tienen un método `reverse` que cambia al array invirtiendo el orden en que aparecen sus elementos. Para este ejercicio, escribe dos funciones, `revertirArray` y `revertirArrayEnSuLugar`. El primero, `revertirArray`, toma un array como argumento y produce un *nuevo* array que tiene los mismos elementos pero en el orden inverso. El segundo, `revertirArrayEnSuLugar`, hace lo que hace el método`reverse`: *modifica* el array dado como argumento invirtiendo sus elementos. Ninguno de los dos puede usar el método `reverse` estándar.

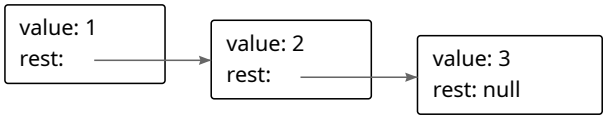
Pensando en las notas acerca de los efectos secundarios y las funciones puras en el capítulo anterior, qué variante esperas que sea útil en más situaciones? Cuál corre más rápido?

Una lista

Los objetos, como conjuntos genéricos de valores, se pueden usar para construir todo tipo de estructuras de datos. Una estructura de datos común es la *lista* (no confundir con un array). Una lista es un conjunto anidado de objetos, con el primer objeto conteniendo una referencia al segundo, el segundo al tercero, y así sucesivamente.

```
let lista = {
  valor: 1,
  resto: {
    valor: 2,
    resto: {
      valor: 3,
      resto: null
    }
  }
};
```

Los objetos resultantes forman una cadena, como esta:



Algo bueno de las listas es que pueden compartir partes de su estructura. Por ejemplo, si creo dos nuevos valores `{valor: 0, resto: lista}` y `{valor: -1, resto: lista}` (con `lista` refiriéndose a la vinculación definida anteriormente), ambos son listas independientes, pero comparten la estructura que conforma sus últimos tres elementos. La lista original también sigue siendo una lista válida de tres elementos.

Escribe una función `arrayALista` que construya una estructura de lista como el que se muestra arriba cuando se le da `[1, 2, 3]` como argumento. También escribe una función `listaAArray` que produzca un array de una lista. Luego agrega una función de utilidad `preceder`, que tome un elemento y una lista y creé una nueva lista que agrega el elemento al frente de la lista de entrada, y `posicion`, que toma una lista y un número y retorne el elemento en la posición dada en la lista (con cero refiriéndose al primer elemento) o `undefined` cuando no exista tal elemento.

Si aún no lo has hecho, también escribe una versión recursiva de `posicion`.

Comparación profunda

El operador `==` compara objetos por identidad. Pero a veces preferirias comparar los valores de sus propiedades reales.

Escribe una función `igualdadProfunda` que toma dos valores y retorne `true` solo si tienen el mismo valor o son objetos con las mismas propiedades, donde los valores de las propiedades sean iguales cuando comparadas con una llamada recursiva a `igualdadProfunda`.

Para saber si los valores deben ser comparados directamente (usa el operador `==` para eso) o si deben tener sus propiedades comparadas, puedes usar el operador `typeof`. Si produce `"object"` para ambos valores, deberías hacer una comparación profunda. Pero tienes que tomar una excepción tonta en cuenta: debido a un accidente histórico, `typeof null` también produce `"object"`.

La función `Object.keys` será útil para cuando necesites revisar las propiedades de los objetos para compararlos.

"Hay dos formas de construir un diseño de software: Una forma es hacerlo tan simple de manera que no hayan deficiencias obvias, y la otra es hacerlo tan complicado de manera que obviamente no hayan deficiencias."

—C.A.R. Hoare, 1980 ACM Turing Award Lecture

Chapter 5

Funciones de Orden Superior

Un programa grande es un programa costoso, y no solo por el tiempo que se necesita para construirlo. El tamaño casi siempre involucra complejidad, y la complejidad confunde a los programadores. A su vez, los programadores confundidos, introducen errores en los programas. Un programa grande entonces proporciona de mucho espacio para que estos bugs se oculten, haciéndolos difíciles de encontrar.

Volvamos rapidamente a los dos últimos programas de ejemplo en la introducción. El primero es auto-contenido y solo tiene seis líneas de largo:

```
let total = 0, cuenta = 1;
while (cuenta <= 10) {
  total += cuenta;
  cuenta += 1;
}
console.log(total);
```

El segundo depende de dos funciones externas y tiene una línea de longitud:

```
console.log(suma(rango(1, 10)));
```

Cuál es más probable que contenga un bug?

Si contamos el tamaño de las definiciones de `suma` y `rango`, el segundo programa también es grande—incluso puede que sea más grande que el primero. Pero aún así, argumentaria que es más probable que sea correcto.

Es más probable que sea correcto porque la solución se expresa en un vocabulario que corresponde al problema que se está resolviendo. Sumar un rango de números no se trata acerca de ciclos y contadores. Se trata acerca de rangos y sumas.

Las definiciones de este vocabulario (las funciones `suma` y `rango`) seguirán involucrando ciclos, contadores y otros detalles incidentales. Pero ya que expresan conceptos más simples que el programa como un conjunto, son más

fáciles de realizar correctamente.

Abstracción

En el contexto de la programación, estos tipos de vocabularios suelen ser llamados *abstracciones*. Las abstracciones esconden detalles y nos dan la capacidad de hablar acerca de los problemas a un nivel superior (o más abstracto).

Como una analogía, compara estas dos recetas de sopa de guisantes:

> Coloque 1 taza de guisantes secos por persona en un recipiente. Agregue agua hasta que los guisantes esten bien cubiertos. Deje los guisantes en agua durante al menos 12 horas. Saque los guisantes del agua y pongalos en una cacerola para cocinar. Agregue 4 tazas de agua por persona. Cubra la sartén y mantenga los guisantes hirviendo a fuego lento durante dos horas. Tome media cebolla por persona. Cortela en piezas con un cuchillo. Agréguela a los guisantes. Tome un tallo de apio por persona. Cortelo en pedazos con un cuchillo. Agréguelo a los guisantes. Tome una zanahoria por persona. Cortela en pedazos. Con un cuchillo! Agregarla a los guisantes. Cocine por 10 minutos más.

Y la segunda receta:

> Por persona: 1 taza de guisantes secos, media cebolla picada, un tallo de apio y una zanahoria.
>
> Remoje los guisantes durante 12 horas. Cocine a fuego lento durante 2 horas en 4 tazas de agua (por persona). Picar y agregar verduras. Cocine por 10 minutos más.

La segunda es más corta y fácil de interpretar. Pero necesitas entender algunas palabras más relacionadas a la cocina—*remojar*, *cocinar a fuego lento*, *picar*, y, supongo, *verduras*.

Cuando programamos, no podemos confiar en que todas las palabras que necesitaremos estaran esperando por nosotros en el diccionario. Por lo tanto, puedes caer en el patrón de la primera receta—resolviendo los pasos precisos que debe realizar la computadora, uno por uno, ciego a los conceptos de orden superior que estos expresan.

En la programación, es una habilidad útil, darse cuenta cuando estás trabajando en un nivel de abstracción demasiado bajo.

Abstrayendo la repetición

Las funciones simples, como las hemos visto hasta ahora, son una buena forma de construir abstracciones. Pero a veces se quedan cortas.

Es común que un programa haga algo una determinada cantidad de veces. Puedes escribir un ciclo `for` para eso, de esta manera:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

Podemos abstraer "hacer algo N veces" como una función? Bueno, es fácil escribir una función que llame a `console.log` N cantidad de veces.

```
function repetirLog(n) {
  for (let i = 0; i < n; i++) {
    console.log(i);
  }
}
```

Pero, y si queremos hacer algo más que loggear los números? Ya que "hacer algo" se puede representar como una función y que las funciones solo son valores, podemos pasar nuestra acción como un valor de función.

```
function repetir(n, accion) {
  for (let i = 0; i < n; i++) {
    accion(i);
  }
}

repetir(3, console.log);
// → 0
// → 1
// → 2
```

No es necesario que le pases una función predefinida a `repetir`. A menudo, desearas crear un valor de función al momento en su lugar.

```
let etiquetas = [];
repetir(5, i => {
  etiquetas.push(`Unidad ${i + 1}`);
});
console.log(etiquetas);
```

```
// → ["Unidad 1", "Unidad 2", "Unidad 3", "Unidad 4", "Unidad 5"]
```

Esto está estructurado un poco como un ciclo `for`—primero describe el tipo de ciclo, y luego provee un cuerpo. Sin embargo, el cuerpo ahora está escrito como un valor de función, que está envuelto en el paréntesis de la llamada a `repetir`. Por eso es que tiene que cerrarse con el corchete de cierre *y* paréntesis de cierre. En casos como este ejemplo, donde el cuerpo es una expresión pequeña y única, podrias tambien omitir las llaves y escribir el ciclo en una sola línea.

Funciones de orden superior

Las funciones que operan en otras funciones, ya sea tomándolas como argumentos o retornandolas, se denominan *funciones de orden superior*. Como ya hemos visto que las funciones son valores regulares, no existe nada particularmente notable sobre el hecho de que tales funciones existen. El término proviene de las matemáticas, donde la distinción entre funciones y otros valores se toma más en serio.

Las funciones de orden superior nos permiten abstraer sobre *acciones*, no solo sobre valores. Estas vienen en varias formas. Por ejemplo, puedes tener funciones que crean nuevas funciones.

```
function mayorQue(n) {
  return m => m > n;
}
let mayorQue10 = mayorQue(10);
console.log(mayorQue10(11));
// → true
```

Y puedes tener funciones que cambien otras funciones.

```
function ruidosa(funcion) {
  return (...argumentos) => {
    console.log("llamando con", argumentos);
    let resultado = funcion(...argumentos);
    console.log("llamada con", argumentos, ", retorno", resultado);
    return resultado;
  };
}
ruidosa(Math.min)(3, 2, 1);
// → llamando con [3, 2, 1]
```

```
// → llamada con [3, 2, 1] , retorno 1
```

Incluso puedes escribir funciones que proporcionen nuevos tipos de flujo de control.

```
function aMenosQue(prueba, entonces) {
  if (!prueba) entonces();
}

repetir(3, n => {
  aMenosQue(n % 2 == 1, () => {
    console.log(n, "es par");
  });
});
// → 0 es par
// → 2 es par
```

Hay un método de array incorporado, `forEach` que proporciona algo como un ciclo `for`/`of` como una función de orden superior.

```
["A", "B"].forEach(letra => console.log(letra));
// → A
// → B
```

Conjunto de datos de códigos

Un área donde brillan las funciones de orden superior es en el procesamiento de datos. Para procesar datos, necesitaremos algunos datos reales. Este capítulo usara un conjunto de datos acerca de códigos—sistema de escrituras como Latin, Cirílico, o Arábico.

Recuerdas Unicode del Capítulo 1, el sistema que asigna un número a cada carácter en el lenguaje escrito. La mayoría de estos carácteres están asociados a un código específico. El estandar contiene 140 codigos diferentes—81 de los cuales todavía están en uso hoy, y 59 que son históricos.

Aunque solo puedo leer con fluidez los caracteres en Latin, aprecio el hecho de que las personas estan escribiendo textos en al menos 80 diferentes sistemas de escritura, muchos de los cuales ni siquiera reconocería. Por ejemplo, aquí está una muestra de escritura a mano en Tamil.

இன்னா செய்தாரை ஒறுத்தல் அவர்நாண
நன்னயம் செய்து விடல்.

El conjunto de datos de ejemplo contiene algunos piezas de información acerca de los 140 codigos definidos en Unicode. Este esta disponible en la caja de arena para este capítulo (*eloquentjavascript.net/code#5*) como la vinculación `SCRIPTS`. La vinculación contiene un array de objetos, cada uno de los cuales describe un codigo.

```
{
  name: "Coptic",
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]],
  direction: "ltr",
  year: -200,
  living: false,
  link: "https://en.wikipedia.org/wiki/Coptic_alphabet"
}
```

Tal objeto te dice el nombre del codigo, los rangos de Unicode asignados a él, la dirección en la que está escrito, la tiempo de origen (aproximado), si todavía está en uso, y un enlace a más información. La dirección en la que esta escrito puede ser `"ltr"` (left-to-right) para izquierda a derecha, `"rtl"` (right-to-left) para derecha a izquierda (la forma en que se escriben los textos en árabe y en hebreo), o `"ttb"` (top-to-bottom) para de arriba a abajo (como con la escritura de Mongolia).

La propiedad `ranges` contiene un array de rangos de caracteres Unicode, cada uno de los cuales es un array de dos elementos que contiene límites inferior y superior. Se asignan los códigos de caracteres dentro de estos rangos al codigo. El limite más bajo es inclusivo (el código 994 es un carácter Copto) y el límite superior es no-inclusivo (el código 1008 no lo es).

Filtrando arrays

Para encontrar los codigos en el conjunto de datos que todavía están en uso, la siguiente función podría ser útil. Filtra hacia afuera los elementos en un array que no pasen una prueba:

```
function filtrar(array, prueba) {
  let pasaron = [];
  for (let elemento of array) {
```

```
    if (prueba(elemento)) {
      pasaron.push(elemento);
    }
  }
  return pasaron;
}

console.log(filtrar(SCRIPTS, codigo => codigo.living));
// → [{name: "Adlam", …}, …]
```

La función usa el argumento llamado `prueba`, un valor de función, para llenar una "brecha" en el cálculo—el proceso de decidir qué elementos recolectar.

Observa cómo la función `filtrar`, en lugar de eliminar elementos del array existente, crea un nuevo array solo con los elementos que pasan la prueba. Esta función es *pura*. No modifica el array que se le es dado.

Al igual que `forEach`, `filtrar` es un método de array estándar, este esta incorporado como `filter`. El ejemplo definió la función solo para mostrar lo que hace internamente. A partir de ahora, la usaremos así en su lugar:

```
console.log(SCRIPTS.filter(codigo => codigo.direction == "ttb"));
// → [{name: "Mongolian", …}, …]
```

Transformando con map

Digamos que tenemos un array de objetos que representan codigos, producidos al filtrar el array `SCRIPTS` de alguna manera. Pero queremos un array de nombres, que es más fácil de inspeccionar

El método `map` ("mapear") transforma un array al aplicar una función a todos sus elementos y construir un nuevo array a partir de los valores retornados. El nuevo array tendrá la misma longitud que el array de entrada, pero su contenido ha sido *mapeado* a una nueva forma en base a la función.

```
function map(array, transformar) {
  let mapeados = [];
  for (let elemento of array) {
    mapeados.push(transformar(elemento));
  }
  return mapeados;
}
```

```
let codigosDerechaAIzquierda = SCRIPTS.filter(codigo => codigo.
   direction == "rtl");
console.log(map(codigosDerechaAIzquierda, codigo => codigo.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", …]
```

Al igual que `forEach` y `filter`, `map` es un método de array estándar.

Resumiendo con reduce

Otra cosa común que hacer con arrays es calcular un valor único a partir de ellos. Nuestro ejemplo recurrente, sumar una colección de números, es una instancia de esto. Otro ejemplo sería encontrar el codigo con la mayor cantidad de caracteres.

La operación de orden superior que representa este patrón se llama *reduce* ("reducir")—a veces también llamada *fold* ("doblar"). Esta construye un valor al repetidamente tomar un solo elemento del array y combinándolo con el valor actual. Al sumar números, comenzarías con el número cero y, para cada elemento, agregas eso a la suma.

Los parámetros para `reduce` son, además del array, una función de combinación y un valor de inicio. Esta función es un poco menos sencilla que `filter` y `map`, así que mira atentamente:

```
function reduce(array, combinar, inicio) {
  let actual = inicio;
  for (let elemento of array) {
    actual = combinar(actual, elemento);
  }
  return actual;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

El método de array estándar `reduce`, que por supuesto corresponde a esta función tiene una mayor comodidad. Si tu array contiene al menos un elemento, tienes permitido omitir el argumento `inicio`. El método tomará el primer elemento del array como su valor de inicio y comienza a reducir a partir del segundo elemento.

```
console.log([1, 2, 3, 4].reduce((a, b) => a + b));
// → 10
```

Para usar `reduce` (dos veces) para encontrar el codigo con la mayor cantidad de caracteres, podemos escribir algo como esto:

```
function cuentaDeCaracteres(codigo) {
  return codigo.ranges.reduce((cuenta, [desde, hasta]) => {
    return cuenta + (hasta - desde);
  }, 0);
}

console.log(SCRIPTS.reduce((a, b) => {
  return cuentaDeCaracteres(a) < cuentaDeCaracteres(b) ? b : a;
}));
// → {name: "Han", …}
```

La función `cuentaDeCaracteres` reduce los rangos asignados a un codigo sumando sus tamaños. Ten en cuenta el uso de la desestructuración en el parámetro lista de la función reductora. La segunda llamada a `reduce` luego usa esto para encontrar el codigo más grande al comparar repetidamente dos scripts y retornando el más grande.

El codigo Han tiene más de 89,000 caracteres asignados en el Estándar Unicode, por lo que es, por mucho, el mayor sistema de escritura en el conjunto de datos. Han es un codigo (a veces) usado para texto chino, japonés y coreano. Esos idiomas comparten muchos caracteres, aunque tienden a escribirlos de manera diferente. El consorcio Unicode (con sede en EE.UU.) decidió tratarlos como un único sistema de escritura para ahorrar códigos de caracteres. Esto se llama *unificación Han* y aún enoja bastante a algunas personas.

Composabilidad

Considera cómo habríamos escrito el ejemplo anterior (encontrar el código más grande) sin funciones de orden superior. El código no es mucho peor.

```
let mayor = null;
for (let codigo of SCRIPTS) {
  if (mayor == null ||
      cuentaDeCaracteres(mayor) < cuentaDeCaracteres(codigo)) {
    mayor = codigo;
  }
}
console.log(mayor);
// → {name: "Han", …}
```

Hay algunos vinculaciones más, y el programa tiene cuatro líneas más. Pero todavía es bastante legible.

Las funciones de orden superior comienzan a brillar cuando necesitas *componer* operaciones. Como ejemplo, vamos a escribir código que encuentre el año de origen promedio para los codigos vivos y muertos en el conjunto de datos.

```
function promedio(array) {
  return array.reduce((a, b) => a + b) / array.length;
}

console.log(Math.round(promedio(
  SCRIPTS.filter(codigo => codigo.living).map(codigo => codigo.year)
    )));
// → 1185
console.log(Math.round(promedio(
  SCRIPTS.filter(codigo => !codigo.living).map(codigo => codigo.year
    ))));
// → 209
```

Entonces, los codigos muertos en Unicode son, en promedio, más antiguos que los vivos. Esta no es una estadística terriblemente significativa o sorprendente. Pero espero que aceptes que el código utilizado para calcularlo no es difícil de leer. Puedes verlo como una tubería: comenzamos con todos los codigos, filtramos los vivos (o muertos), tomamos los años de aquellos, los promediamos, y redondeamos el resultado.

Definitivamente también podrías haber escribir este codigo como un gran ciclo.

```
let total = 0, cuenta = 0;
for (let codigo of SCRIPTS) {
  if (codigo.living) {
    total += codigo.year;
    cuenta += 1;
  }
}
console.log(Math.round(total / cuenta));
// → 1185
```

Pero es más difícil de ver qué se está calculando y cómo. Y ya que los resultados intermedios no se representan como valores coherentes, sería mucho más trabajo extraer algo así como `promedio` en una función aparte.

En términos de lo que la computadora realmente está haciendo, estos dos enfoques también son bastante diferentes. El primero creará nuevos arrays al ejecutar `filter` y `map`, mientras que el segundo solo computa algunos números, haciendo menos trabajo. Por lo general, puedes permitirte el enfoque legible, pero si estás procesando arrays enormes, y haciendolo muchas veces, el estilo menos abstracto podría ser mejor debido a la velocidad extra.

Strings y códigos de caracteres

Un uso del conjunto de datos sería averiguar qué código esta usando una pieza de texto. Veamos un programa que hace esto.

Recuerda que cada codigo tiene un array de rangos para los códigos de caracteres asociados a el. Entonces, dado un código de carácter, podríamos usar una función como esta para encontrar el codigo correspondiente (si lo hay):

```
function codigoCaracter(codigo_caracter) {
  for (let codigo of SCRIPTS) {
    if (codigo.ranges.some(([desde, hasta]) => {
      return codigo_caracter >= desde && codigo_caracter < hasta;
    })) {
      return codigo;
    }
  }
  return null;
}

console.log(codigoCaracter(121));
// → {name: "Latin", …}
```

El método `some` (“alguno”) es otra función de orden superior. Toma una función de prueba y te dice si esa función retorna verdadero para cualquiera de los elementos en el array.

Pero cómo obtenemos los códigos de los caracteres en un string?

En el Capítulo 1 mencioné que los strings de JavaScript estan codificados como una secuencia de números de 16 bits. Estos se llaman *unidades de código*. Inicialmente se suponía que un código de carácter Unicode encajara dentro de esa unidad (lo que da un poco más de 65,000 caracteres). Cuando quedó claro que esto no seria suficiente, muchas personas se resistieron a la necesidad de usar más memoria por carácter. Para apaciguar estas preocupaciones, UTF-16, el formato utilizado por los strings de JavaScript, fue inventado. Este describe la mayoría de los caracteres mas comunes usando una sola unidad de código de

16 bits, pero usa un par de dos de esas unidades para otros caracteres.

Al dia de hoy UTF-16 generalmente se considera como una mala idea. Parece casi intencionalmente diseñado para invitar a errores. Es fácil escribir programas que pretenden que las unidades de código y caracteres son la misma cosa. Y si tu lenguaje no usa caracteres de dos unidades, esto parecerá funcionar simplemente bien. Pero tan pronto como alguien intente usar dicho programa con algunos menos comunes caracteres chinos, este se rompe. Afortunadamente, con la llegada del emoji, todo el mundo ha empezado a usar caracteres de dos unidades, y la carga de lidiar con tales problemas esta bastante mejor distribuida.

Desafortunadamente, las operaciones obvias con strings de JavaScript, como obtener su longitud a través de la propiedad `length` y acceder a su contenido usando corchetes, trata solo con unidades de código.

```
// Dos caracteres emoji, caballo y zapato
let caballoZapato = "🏇👟";
console.log(caballoZapato.length);
// → 4
console.log(caballoZapato[0]);
// → ((Medio-carácter inválido))
console.log(caballoZapato.charCodeAt(0));
// → 55357 (Código del medio-carácter)
console.log(caballoZapato.codePointAt(0));
// → 128052 (Código real para emoji de caballo)
```

El método `charCodeAt` de JavaScript te da una unidad de código, no un código de carácter completo. El método `codePointAt`, añadido despues, si da un carácter completo de Unicode. Entonces podríamos usarlo para obtener caracteres de un string. Pero el argumento pasado a `codePointAt` sigue siendo un índice en la secuencia de unidades de código. Entonces, para hacer un ciclo a traves de todos los caracteres en un string, todavía tendríamos que lidiar con la cuestión de si un carácter ocupa una o dos unidades de código.

En el capítulo anterior, mencioné que el ciclo `for/of` también se puede usar en strings. Como `codePointAt`, este tipo de ciclo se introdujo en un momento en que las personas eran muy conscientes de los problemas con UTF-16. Cuando lo usas para hacer un ciclo a traves de un string, te da caracteres reales, no unidades de código.

```
let dragonRosa = "🐉🌹";
for (let caracter of dragonRosa) {
  console.log(caracter);
```

```
}
// → 🐉
// → 🌹
```

Si tienes un caracter (que será un string de unidades de uno o dos códigos), puedes usar `codePointAt(0)` para obtener su código.

Reconociendo texto

Tenemos una función `codigoCaracter` y una forma de correctamente hacer un ciclo a traves de caracteres. El siguiente paso sería contar los caracteres que pertenecen a cada codigo. La siguiente abstracción de conteo será útil para eso:

```
function contarPor(elementos, nombreGrupo) {
  let cuentas = [];
  for (let elemento of elementos) {
    let nombre = nombreGrupo(elemento);
    let conocido = cuentas.findIndex(c => c.nombre == nombre);
    if (conocido == -1) {
      cuentas.push({nombre, cuenta: 1});
    } else {
      cuentas[conocido].cuenta++;
    }
  }
  return cuentas;
}

console.log(contarPor([1, 2, 3, 4, 5], n => n > 2));
// → [{nombre: false, cuenta: 2}, {nombre: true, cuenta: 3}]
```

La función `contarPor` espera una colección (cualquier cosa con la que podamos hacer un ciclo `for/of`) y una función que calcula un nombre de grupo para un elemento dado. Retorna un array de objetos, cada uno de los cuales nombra un grupo y te dice la cantidad de elementos que se encontraron en ese grupo.

Utiliza otro método de array—`findIndex` ("encontrar index"). Este método es algo así como `indexOf`, pero en lugar de buscar un valor específico, este encuentra el primer valor para el cual la función dada retorna verdadero. Como `indexOf`, retorna -1 cuando no se encuentra dicho elemento.

Usando `contarPor`, podemos escribir la función que nos dice qué codigos se usan en una pieza de texto.

```
function codigosTexto(texto) {
  let codigos = contarPor(texto, caracter => {
    let codigo = codigoCaracter(caracter.codePointAt(0));
    return codigo ? codigo.name : "ninguno";
  }).filter(({name}) => name != "ninguno");

  let total = codigos.reduce((n, {count}) => n + count, 0);
  if (total == 0) return "No se encontraron codigos";

  return codigos.map(({name, count}) => {
    return `${Math.round(count * 100 / total)}% ${name}`;
  }).join(", ");
}

console.log(codigosTexto('英国的狗说"woof", 俄罗斯的狗说"тяв"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

La función primero cuenta los caracteres por nombre, usando `codigoCaracter` para asignarles un nombre, y recurre al string `"ninguno"` para caracteres que no son parte de ningún codigo. La llamada `filter` deja afuera las entrada para `"ninguno"` del array resultante, ya que no estamos interesados en esos caracteres.

Para poder calcular porcentajes, primero necesitamos la cantidad total de caracteres que pertenecen a un codigo, lo que podemos calcular con `reduce`. Si no se encuentran tales caracteres, la función retorna un string específico. De lo contrario, transforma las entradas de conteo en strings legibles con `map` y luego las combina con `join`.

Resumen

Ser capaz de pasar valores de función a otras funciones es un aspecto profundamente útil de JavaScript. Nos permite escribir funciones que modelen calculos con "brechas" en ellas. El código que llama a estas funciones pueden llenar estas brechas al proporcionar valores de función.

Los arrays proporcionan varios métodos útiles de orden superior. Puedes usar `forEach` para recorrer los elementos en un array. El método `filter` retorna un nuevo array que contiene solo los elementos que pasan una función de predicado. Transformar un array al poner cada elemento a través de una función se hace con `map`. Puedes usar `reduce` para combinar todos los elementos en una array a un solo valor. El método `some` prueba si algun elemento coincide con una función de predicado determinada. Y `findIndex` encuentra la posición

del primer elemento que coincide con un predicado.

Ejercicios

Aplanamiento

Use el método `reduce` en combinación con el método `concat` para "aplanar" un array de arrays en un único array que tenga todos los elementos de los arrays originales.

Tu propio ciclo

Escriba una función de orden superior llamada `ciclo` que proporcione algo así como una declaración de ciclo `for`. Esta toma un valor, una función de prueba, una función de actualización y un cuerpo de función. En cada iteración, primero ejecuta la función de prueba en el valor actual del ciclo y se detiene si esta retorna falso. Luego llama al cuerpo de función, dándole el valor actual. Y finalmente, llama a la función de actualización para crear un nuevo valor y comienza desde el principio.

Cuando definas la función, puedes usar un ciclo regular para hacer los ciclos reales.

Cada

De forma análoga al método `some`, los arrays también tienen un método `every` ("cada"). Este retorna true cuando la función dada devuelve verdadero para *cada* elemento en el array. En cierto modo, `some` es una versión del operador `||` que actúa en arrays, y `every` es como el operador `&&`.

Implementa `every` como una función que tome un array y una función predicado como parámetros. Escribe dos versiones, una usando un ciclo y una usando el método `some`.

Dirección de Escritura Dominante

Escriba una función que calcule la dirección de escritura dominante en un string de texto. Recuerde que cada objeto de codigo tiene una propiedad `direction` que puede ser `"ltr"` (de izquierda a derecha), `"rtl"` (de derecha a izquierda), o `"ttb"` (arriba a abajo).

La dirección dominante es la dirección de la mayoría de los caracteres que tienen un código asociado a ellos. Las funciones `codigoCaracter` y `contarPor` definidas anteriormente en el capítulo probablemente seran útiles aquí.

"Un tipo de datos abstracto se realiza al escribir un tipo especial de programa [...] que define el tipo en base a las operaciones que puedan ser realizadas en él."

—Barbara Liskov, Programming with Abstract Data Types

Chapter 6

La Vida Secreta de los Objetos

El Capítulo 4 introdujo los objetos en JavaScript. En la cultura de la programación, tenemos una cosa llamada *programación orientada a objetos*, la cual es un conjunto de técnicas que usan objetos (y conceptos relacionados) como el principio central de la organización del programa.

Aunque nadie realmente está de acuerdo con su definición exacta, la programación orientada a objetos ha contribuido al diseño de muchos lenguajes de programación, incluyendo JavaScript. Este capítulo describirá la forma en la que estas ideas pueden ser aplicadas en JavaScript.

Encapsulación

La idea central en la programación orientada a objetos es dividir a los programas en piezas más pequeñas y hacer que cada pieza sea responsable de gestionar su propio estado.

De esta forma, los conocimientos acerca de como funciona una parte del programa pueden mantenerse *locales* a esa pieza. Alguien trabajando en otra parte del programa no tiene que recordar o ni siquiera tener una idea de ese conocimiento. Cada vez que los detalles locales cambien, solo el código directamente a su alrededor debe ser actualizado.

Las diferentes piezas de un programa como tal, interactúan entre sí a través de *interfaces*, las cuales son conjuntos limitados de funciones y vinculaciones que proporcionan funcionalidades útiles en un nivel más abstracto, ocultando asi su implementación interna.

Tales piezas del programa se modelan usando objetos. Sus interfaces consisten en un conjunto específico de métodos y propiedades. Las propiedades que son parte de la interfaz se llaman *publicas*. Las otras, las cuales no deberian ser tocadas por el código externo , se les llama *privadas*.

Muchos lenguajes proporcionan una forma de distinguir entre propiedades publicas y privadas, y ademas evitarán que el código externo pueda acceder a las privadas por completo. JavaScript, una vez más tomando el enfoque

minimalista, no hace esto. Todavía no, al menos—hay trabajo en camino para agregar esto al lenguaje.

Aunque el lenguaje no tenga esta distinción incorporada, los programadores de JavaScript *estan* usando esta idea con éxito .Típicamente, la interfaz disponible se describe en la documentación o en los comentarios. También es común poner un carácter de guión bajo (_) al comienzo de los nombres de las propiedades para indicar que estas propiedades son privadas.

Separar la interfaz de la implementación es una gran idea. Esto usualmente es llamado *encapsulación*.

MÉTODOS

Los métodos no son más que propiedades que tienen valores de función. Este es un método simple:

```
let conejo = {};
conejo.hablar = function(linea) {
  console.log(`El conejo dice '${linea}'`);
};

conejo.hablar("Estoy vivo.");
// → El conejo dice 'Estoy vivo.'
```

Por lo general, un método debe hacer algo en el objeto con que se llamó. Cuando una función es llamada como un método—buscada como una propiedad y llamada inmediatamente, como en `objeto.metodo()`—la vinculación llamada `this` ("este") en su cuerpo apunta automáticamente al objeto en la cual fue llamada.

```
function hablar(linea) {
  console.log(`El conejo ${this.tipo} dice '${linea}'`);
}
let conejoBlanco = {tipo: "blanco", hablar};
let conejoHambriento = {tipo: "hambriento", hablar};

conejoBlanco.hablar("Oh mis orejas y bigotes, " +
                    "que tarde se esta haciendo!");
// → El conejo blanco dice 'Oh mis orejas y bigotes, que
//   tarde se esta haciendo!'
conejoHambriento.hablar("Podria comerme una zanahoria ahora mismo.")
    ;
// → El conejo hambriento dice 'Podria comerme una zanahoria ahora
```

```
    mismo.'
```

Puedes pensar en `this` como un parámetro extra que es pasado en una manera diferente. Si quieres pasarlo explícitamente, puedes usar el método `call` ("llamar") de una función, que toma el valor de `this` como primer argumento y trata a los argumentos adicionales como parámetros normales.

```
hablar.call(conejoHambriento, "Burp!");
// → El conejo hambriento dice 'Burp!'
```

Como cada función tiene su propia vinculación `this`, cuyo valor depende de la forma en como esta se llama, no puedes hacer referencia al `this` del alcance envolvente en una función regular definida con la palabra clave `function`.

Las funciones de flecha son diferentes—no crean su propia vinculación `this`, pero pueden ver la vinculación`this` del alcance a su alrededor. Por lo tanto, puedes hacer algo como el siguiente código, que hace referencia a `this` desde adentro de una función local:

```
function normalizar() {
  console.log(this.coordinadas.map(n => n / this.length));
}
normalizar.call({coordinadas: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

Si hubieras escrito el argumento para `map` usando la palabra clave `function`, el código no funcionaría.

Prototipos

Observa atentamente.

```
let vacio = {};
console.log(vacio.toString);
// → function toString()…{}
console.log(vacio.toString());
// → [object Object]
```

Saqué una propiedad de un objeto vacío. Magia!

Bueno, en realidad no. Simplemente he estado ocultando información acerca de como funcionan los objetos en JavaScript. En adición a su conjunto de

propiedades, la mayoría de los objetos también tienen un *prototipo*. Un prototipo es otro objeto que se utiliza como una reserva de propiedades alternativa. Cuando un objeto recibe una solicitud por una propiedad que este no tiene, se buscará en su prototipo la propiedad, luego en el prototipo del prototipo y asi sucesivamente.

Asi que, quién es el prototipo de ese objeto vacío? Es el gran prototipo ancestral, la entidad detrás de casi todos los objetos, `Object.prototype` ("Objeto.prototipo").

```
console.log(Object.getPrototypeOf({}) ==
            Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

Como puedes adivinar, `Object.getPrototypeOf` ("Objeto.obtenerPrototipoDe") retorna el prototipo de un objeto.

Las relaciones prototipo de los objetos en JavaScript forman una estructura en forma de árbol, y en la raíz de esta estructura se encuentra `Object.prototype` . Este proporciona algunos métodos que pueden ser accedidos por todos los objetos, como `toString`, que convierte un objeto en una representación de tipo string.

Muchos objetos no tienen `Object.prototype` directamente como su prototipo, pero en su lugar tienen otro objeto que proporciona un conjunto diferente de propiedades predeterminadas. Las funciones derivan de `Function.prototype`, y los arrays derivan de `Array.prototype`.

```
console.log(Object.getPrototypeOf(Math.max) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
            Array.prototype);
// → true
```

Tal prototipo de objeto tendrá en si mismo un prototipo, a menudo `Object.prototype`, por lo que aún proporciona indirectamente métodos como `toString`.

Puede usar `Object.create` para crear un objeto con un prototipo especifico.

```
let conejoPrototipo = {
  hablar(linea) {
    console.log(`El conejo ${this.tipo} dice '${linea}'`);
```

```
  }
};
let conejoAsesino = Object.create(conejoPrototipo);
conejoAsesino.tipo = "asesino";
conejoAsesino.hablar("SKREEEE!");
// → El conejo asesino dice 'SKREEEE!'
```

Una propiedad como `hablar(linea)` en una expresión de objeto es un atajo para definir un método. Esta crea una propiedad llamada `hablar` y le da una función como su valor.

El conejo "prototipo" actúa como un contenedor para las propiedades que son compartidas por todos los conejos. Un objeto de conejo individual, como el conejo asesino, contiene propiedades que aplican solo a sí mismo—en este caso su tipo—y deriva propiedades compartidas desde su prototipo.

Clases

El sistema de prototipos en JavaScript se puede interpretar como un enfoque informal de un concepto orientado a objetos llamado *clasees*. Una clase define la forma de un tipo de objeto—qué métodos y propiedades tiene este. Tal objeto es llamado una *instancia* de la clase.

Los prototipos son útiles para definir propiedades en las cuales todas las instancias de una clase compartan el mismo valor, como métodos. Las propiedades que difieren por instancia, como la propiedad `tipo` en nuestros conejos, necesitan almacenarse directamente en los objetos mismos.

Entonces, para crear una instancia de una clase dada, debes crear un objeto que derive del prototipo adecuado, pero *también* debes asegurarte de que, en sí mismo, este objeto tenga las propiedades que las instancias de esta clase se supone que tengan. Esto es lo que una función *constructora* hace.

```
function crearConejo(tipo) {
  let conejo = Object.create(conejoPrototipo);
  conejo.tipo = tipo;
  return conejo;
}
```

JavaScript proporciona una manera de hacer que la definición de este tipo de funciones sea más fácil. Si colocas la palabra clave `new` ("new") delante de una llamada de función, la función sera tratada como un constructor. Esto significa que un objeto con el prototipo adecuado es creado automáticamente,

vinculado a `this` en la función, y retornado al final de la función.

El objeto prototipo utilizado al construir objetos se encuentra al tomar la propiedad `prototype` de la función constructora.

```
function Conejo(tipo) {
  this.tipo = tipo;
}
Conejo.prototype.hablar = function(linea) {
  console.log(`El conejo ${this.tipo} dice '${linea}'`);
};

let conejoRaro = new Conejo("raro");
```

Los constructores (todas las funciones, de hecho) automáticamente obtienen una propiedad llamada `prototype`, que por defecto contiene un objeto simple y vacío, que deriva de `Object.prototype`. Puedes sobrescribirlo con un nuevo objeto si asi quieres. O puedes agregar propiedades al objeto ya existente, como lo hace el ejemplo.

Por convención, los nombres de los constructores tienen la primera letra en mayúscula para que se puedan distinguir fácilmente de otras funciones.

Es importante entender la distinción entre la forma en que un prototipo está asociado con un constructor (a través de su propiedad `prototype`) y la forma en que los objetos *tienen* un prototipo (que se puede encontrar con `Object.getPrototypeOf`). El prototipo real de un constructor es `Function.prototype`, ya que los constructores son funciones. Su *propiedad* `prototype` contiene el prototipo utilizado para las instancias creadas a traves de el.

```
console.log(Object.getPrototypeOf(Conejo) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf(conejoRaro) ==
            Conejo.prototype);
// → true
```

Notación de clase

Entonces, las clasees en JavaScript son funciones constructoras con una propiedad prototipo. Así es como funcionan, y hasta 2015, esa era la manera en como tenías que escribirlas. Estos días, tenemos una notación menos incómoda.

```
class Conejo {
  constructor(tipo) {
    this.tipo = tipo;
  }
  hablar(linea) {
    console.log(`El conejo ${this.tipo} dice '${linea}'`);
  }
}

let conejoAsesino = new Conejo("asesino");
let conejoNegro = new Conejo("negro");
```

La palabra clave `class` ("clase") comienza una declaración de clase, que nos permite definir un constructor y un conjunto de métodos, todo en un solo lugar. Cualquier número de métodos se pueden escribir dentro de las llaves de la declaración. El metodo llamado `constructor` es tratado de una manera especial. Este proporciona la función constructora real, que estará vinculada al nombre `Conejo`. Los otros metodos estaran empacados en el prototipo de ese constructor. Por lo tanto, la declaración de clase anterior es equivalente a la definición de constructor en la sección anterior. Solo que se ve mejor.

Actualmente las declaraciones de clase solo permiten que los *metodos*—propiedades que contengan funciones—puedan ser agregados al prototipo. Esto puede ser algo inconveniente para cuando quieras guardar un valor no-funcional allí. La próxima versión del lenguaje probablemente mejore esto. Por ahora, tú puedes crear tales propiedades al manipular directamente el prototipo después de haber definido la clase.

Al igual que `function`, `class` se puede usar tanto en posiciones de declaración como de expresión. Cuando se usa como una expresión, no define una vinculación, pero solo produce el constructor como un valor. Tienes permitido omitir el nombre de clase en una expresión de clase.

```
let objeto = new class { obtenerPalabra() { return "hola"; } };
console.log(objeto.obtenerPalabra());
// → hola
```

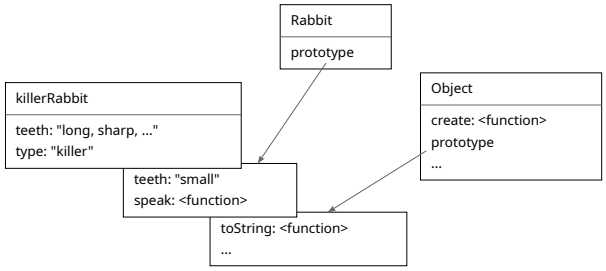
Sobreescribiendo propiedades derivadas

Cuando le agregas una propiedad a un objeto, ya sea que esté presente en el prototipo o no, la propiedad es agregada al objeto *en si mismo*. Si ya había una

propiedad con el mismo nombre en el prototipo, esta propiedad ya no afectará al objeto, ya que ahora está oculta detrás de la propiedad del propio objeto.

```
Rabbit.prototype.dientes = "pequeños";
console.log(conejoAsesino.dientes);
// → pequeños
conejoAsesino.dientes = "largos, filosos, y sangrientos";
console.log(conejoAsesino.dientes);
// → largos, filosos, y sangrientos
console.log(conejoNegro.dientes);
// → pequeños
console.log(Rabbit.prototype.dientes);
// → pequeños
```

El siguiente diagrama esboza la situación después de que este código ha sido ejecutado. Los prototipos de `Conejo` y `Object` se encuentran detrás de `conejoAsesino` como una especie de telón de fondo, donde las propiedades que no se encuentren en el objeto en sí mismo puedan ser buscadas.



Sobreescribir propiedades que existen en un prototipo puede ser algo útil que hacer. Como muestra el ejemplo de los dientes de conejo, esto se puede usar para expresar propiedades excepcionales en instancias de una clase más genérica de objetos, dejando que los objetos no-excepcionales tomen un valor estándar desde su prototipo.

También puedes sobreescribir para darle a los prototipos estándar de función y array un método diferente `toString` al del objeto prototipo básico.

```
console.log(Array.prototype.toString ==
            Object.prototype.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

Llamar a `toString` en un array da un resultado similar al de una llamada

`.join(",")` en él—pone comas entre los valores del array. Llamar directamente a `Object.prototype.toString` con un array produce un string diferente. Esa función no sabe acerca de los arrays, por lo que simplemente pone la palabra *object* y el nombre del tipo entre corchetes.

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

Mapas

Vimos a la palabra *map* usada en el capítulo anterior para una operación que transforma una estructura de datos al aplicar una función en sus elementos.

Un *mapa* (sustantivo) es una estructura de datos que asocia valores (las llaves) con otros valores. Por ejemplo, es posible que desees mapear nombres a edades. Es posible usar objetos para esto.

```
let edades = {
  Boris: 39,
  Liang: 22,
  Júlia: 62
};

console.log(`Júlia tiene ${edades["Júlia"]}`);
// → Júlia tiene 62
console.log("Se conoce la edad de Jack?", "Jack" in edades);
// → Se conoce la edad de Jack? false
console.log("Se conoce la edad de toString?", "toString" in edades);
// → Se conoce la edad de toString? true
```

Aquí, los nombres de las propiedades del objeto son los nombres de las personas, y los valores de las propiedades sus edades. Pero ciertamente no incluimos a nadie llamado toString en nuestro mapa. Sin embargo, debido a que los objetos simples se derivan de `Object.prototype`, parece que la propiedad está ahí.

Como tal, usar objetos simples como mapas es peligroso. Hay varias formas posibles de evitar este problema. Primero, es posible crear objetos sin *ningun* prototipo. Si pasas `null` a `Object.create`, el objeto resultante no se derivará de `Object.prototype` y podra ser usado de forma segura como un mapa.

```
console.log("toString" in Object.create(null));
```

```
// → false
```

Los nombres de las propiedades de los objetos deben ser strings. Si necesitas un mapa cuyas claves no puedan ser convertidas fácilmente a strings—como objetos—no puedes usar un objeto como tu mapa.

Afortunadamente, JavaScript viene con una clase llamada `Map` que esta escrita para este propósito exacto. Esta almacena un mapeo y permite cualquier tipo de llaves.

```
let edades = new Map();
edades.set("Boris", 39);
edades.set("Liang", 22);
edades.set("Júlia", 62);

console.log(`Júlia tiene ${edades.get("Júlia")}`);
// → Júlia tiene 62
console.log("Se conoce la edad de Jack?", edades.has("Jack"));
// → Se conoce la edad de Jack? false
console.log(edades.has("toString"));
// → false
```

Los métodos `set` (“establecer”),`get` (“obtener”), y `has` (“tiene”) son parte de la interfaz del objeto `Map`. Escribir una estructura de datos que pueda actualizarse rápidamente y buscar en un gran conjunto de valores no es fácil, pero no tenemos que preocuparnos acerca de eso. Alguien más lo hizo por nosotros, y podemos utilizar esta simple interfaz para usar su trabajo.

Si tienes un objeto simple que necesitas tratar como un mapa por alguna razón, es útil saber que `Object.keys` solo retorna las llaves propias del objeto, no las que estan en el prototipo. Como alternativa al operador `in`, puedes usar el método`hasOwnProperty` (“tienePropiaPropiedad”), el cual ignora el prototipo del objeto.

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

Polimorfismo

Cuando llamas a la función `String` (que convierte un valor a un string) en un objeto, llamará al método `toString` en ese objeto para tratar de crear un string significativo a partir de el. Mencioné que algunos de los prototipos estándar definen su propia versión de `toString` para que puedan crear un string que contenga información más útil que `"[object Object]"`. También puedes hacer eso tú mismo.

```
Conejo.prototype.toString = function() {
  return `un conejo ${this.tipo}`;
};

console.log(String(conejoNegro));
// → un conejo negro
```

Esta es una instancia simple de una idea poderosa. Cuando un pedazo de código es escrito para funcionar con objetos que tienen una cierta interfaz—en este caso, un método `toString`—cualquier tipo de objeto que soporte esta interfaz se puede conectar al código, y simplemente funcionará.

Esta técnica se llama *polimorfismo*. El código polimórfico puede funcionar con valores de diferentes formas, siempre y cuando soporten la interfaz que este espera.

Mencioné en el Capítulo 4 que un ciclo `for/of` puede recorrer varios tipos de estructuras de datos. Este es otro caso de polimorfismo—tales ciclos esperan que la estructura de datos exponga una interfaz específica, lo que hacen los arrays y strings. Y también puedes agregar esta interfaz a tus propios objetos! Pero antes de que podamos hacer eso, necesitamos saber qué son los símbolos.

Símbolos

Es posible que múltiples interfaces usen el mismo nombre de propiedad para diferentes cosas. Por ejemplo, podría definir una interfaz en la que se suponga que el método `toString` convierte el objeto a una pieza de hilo. No sería posible para un objeto ajustarse a esa interfaz y al uso estándar de `toString`.

Esa sería una mala idea, y este problema no es muy común. La mayoria de los programadores de JavaScript simplemente no piensan en eso. Pero los diseñadores del lenguaje, cuyo *trabajo* es pensar acerca de estas cosas, nos han proporcionado una solución de todos modos.

Cuando afirmé que los nombres de propiedad son strings, eso no fue del todo

preciso. Usualmente lo son, pero también pueden ser *símbolos*. Los símbolos son valores creados con la función `Symbol`. A diferencia de los strings, los símbolos recién creados son únicos—no puedes crear el mismo símbolo dos veces.

```
let simbolo = Symbol("nombre");
console.log(simbolo == Symbol("nombre"));
// → false
Conejo.prototype[simbolo] = 55;
console.log(conejoNegro[simbolo]);
// → 55
```

El string que pases a `Symbol` es incluido cuando lo conviertas a string, y puede hacer que sea más fácil reconocer un símbolo cuando, por ejemplo, lo muestres en la consola. Pero no tiene sentido más allá de eso—múltiples símbolos pueden tener el mismo nombre.

Al ser únicos y utilizables como nombres de propiedad, los símbolos son adecuados para definir interfaces que pueden vivir pacíficamente junto a otras propiedades, sin importar cuáles sean sus nombres.

```
const simboloToString = Symbol("toString");
Array.prototype[simboloToString] = function() {
  return `${this.length} cm de hilo azul`;
};

console.log([1, 2].toString());
// → 1,2
console.log([1, 2][simboloToString]());
// → 2 cm de hilo azul
```

Es posible incluir propiedades de símbolos en expresiones de objetos y clases usando corchetes alrededor del nombre de la propiedad. Eso hace que se evalúe el nombre de la propiedad, al igual que la notación de corchetes para acceder propiedades, lo cual nos permite hacer referencia a una vinculación que contiene el símbolo.

```
let objetoString = {
  [simboloToString]() { return "una cuerda de cañamo"; }
};
console.log(objetoString[simboloToString]());
// → una cuerda de cañamo
```

La interfaz de iterador

Se espera que el objeto dado a un ciclo `for/of` sea *iterable*. Esto significa que tenga un método llamado con el símbolo `Symbol.iterator` (un valor de símbolo definido por el idioma, almacenado como una propiedad de la función `Symbol`).

Cuando sea llamado, ese método debe retornar un objeto que proporcione una segunda interfaz, *iteradora*. Esta es la cosa real que realiza la iteración. Tiene un método `next` ("siguiente") que retorna el siguiente resultado. Ese resultado debería ser un objeto con una propiedad `value` ("valor"), que proporciona el siguiente valor, si hay uno, y una propiedad `done` ("listo") que debería ser cierta cuando no haya más resultados y falso de lo contrario.

Ten en cuenta que los nombres de las propiedades `next`, `value` y `done` son simples strings, no símbolos. Solo `Symbol.iterator`, que probablemente sea agregado a un *monton* de objetos diferentes, es un símbolo real.

Podemos usar directamente esta interfaz nosotros mismos.

```
let iteradorOK = "OK"[Symbol.iterator]();
console.log(iteradorOK.next());
// → {value: "O", done: false}
console.log(iteradorOK.next());
// → {value: "K", done: false}
console.log(iteradorOK.next());
// → {value: undefined, done: true}
```

Implementemos una estructura de datos iterable. Construiremos una clase *matriz*, que actuara como un array bidimensional.

```
class Matriz {
  constructor(ancho, altura, elemento = (x, y) => undefined) {
    this.ancho = ancho;
    this.altura = altura;
    this.contenido = [];

    for (let y = 0; y < altura; y++) {
      for (let x = 0; x < ancho; x++) {
        this.contenido[y * ancho + x] = elemento(x, y);
      }
    }
  }

  obtener(x, y) {
    return this.contenido[y * this.ancho + x];
  }
```

```
  establecer(x, y, valor) {
    this.contenido[y * this.ancho + x] = valor;
  }
}
```

La clase almacena su contenido en un único array de elementos *altura* × *ancho*. Los elementos se almacenan fila por fila, por lo que, por ejemplo, el tercer elemento en la quinta fila es (utilizando indexación basada en cero) almacenado en la posición $4 \times ancho + 2$.

La función constructora toma un ancho, una altura y una función opcional de contenido que se usará para llenar los valores iniciales. Hay métodos `obtener` y `establecer` para recuperar y actualizar elementos en la matriz.

Al hacer un ciclo sobre una matriz, generalmente estás interesado en la posición tanto de los elementos como de los elementos en sí mismos, así que haremos que nuestro iterador produzca objetos con propiedades `x`, `y`, y `value` ("valor").

```
class IteradorMatriz {
  constructor(matriz) {
    this.x = 0;
    this.y = 0;
    this.matriz = matriz;
  }

  next() {
    if (this.y == this.matriz.altura) return {done: true};

    let value = {x: this.x,
                 y: this.y,
                 value: this.matriz.obtener(this.x, this.y)};
    this.x++;
    if (this.x == this.matriz.ancho) {
      this.x = 0;
      this.y++;
    }
    return {value, done: false};
  }
}
```

La clase hace un seguimiento del progreso de iterar sobre una matriz en sus propiedades `x` y `y`. El método `next` ("siguiente") comienza comprobando si la parte inferior de la matriz ha sido alcanzada. Si no es así, *primero* crea el objeto que contiene el valor actual y *luego* actualiza su posición, moviéndose a

la siguiente fila si es necesario.

Configuremos la clase `Matriz` para que sea iterable. A lo largo de este libro, Ocasionalmente usaré la manipulación del prototipo después de los hechos para agregar métodos a clases, para que las piezas individuales de código permanezcan pequeñas y autónomas. En un programa regular, donde no hay necesidad de dividir el código en pedazos pequeños, declararias estos métodos directamente en la clase.

```
Matriz.prototype[Symbol.iterator] = function() {
  return new IteradorMatriz(this);
};
```

Ahora podemos recorrer una matriz con `for/of`.

```
let matriz = new Matriz(2, 2, (x, y) => `valor ${x},${y}`);
for (let {x, y, value} of matriz) {
  console.log(x, y, value);
}
// → 0 0 valor 0,0
// → 1 0 valor 1,0
// → 0 1 valor 0,1
// → 1 1 valor 1,1
```

Getters, setters y estáticos

A menudo, las interfaces consisten principalmente de métodos, pero también está bien incluir propiedades que contengan valores que no sean de función. Por ejemplo, los objetos `Map` tienen una propiedad `size` ("tamaño") que te dice cuántas claves hay almacenanadas en ellos.

Ni siquiera es necesario que dicho objeto calcule y almacene tales propiedades directamente en la instancia. Incluso las propiedades que pueden ser accedidas directamente pueden ocultar una llamada a un método. Tales métodos se llaman *getters*, y se definen escribiendo `get` ("obtener") delante del nombre del método en una expresión de objeto o declaración de clase.

```
let tamañoCambiante = {
  get tamaño() {
    return Math.floor(Math.random() * 100);
  }
};
```

```
console.log(tamañoCambiante.tamaño);
// → 73
console.log(tamañoCambiante.tamaño);
// → 49
```

Cuando alguien lee desde la propiedad `tamaño` de este objeto, el método asociado es llamado. Puedes hacer algo similar cuando se escribe en una propiedad, usando un *setter*.

```
class Temperatura {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(valor) {
    this.celsius = (valor - 32) / 1.8;
  }

  static desdeFahrenheit(valor) {
    return new Temperatura((valor - 32) / 1.8);
  }
}

let temp = new Temperatura(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

La clase `Temperatura` te permite leer y escribir la temperatura ya sea en grados Celsius o grados Fahrenheit, pero internamente solo almacena Celsius y convierte automáticamente a Celsius en el getter y setter `fahrenheit`.

Algunas veces quieres adjuntar algunas propiedades directamente a tu función constructora, en lugar de al prototipo. Tales métodos no tienen acceso a una instancia de clase, pero pueden, por ejemplo, ser utilizados para proporcionar formas adicionales de crear instancias.

Dentro de una declaración de clase, métodos que tienen `static` ("estatico") escrito antes su nombre son almacenados en el constructor. Entonces, la clase `Temperatura` te permite escribir `Temperature.desdeFahrenheit(100)` para crear

una temperatura usando grados Fahrenheit.

Herencia

Algunas matrices son conocidas por ser *simétricas*. Si duplicas una matriz simétrico alrededor de su diagonal de arriba-izquierda a derecha-abajo, esta se mantiene igual. En otras palabras, el valor almacenado en *x*,*y* es siempre el mismo al de *y*,*x*.

Imagina que necesitamos una estructura de datos como `Matriz` pero que haga cumplir el hecho de que la matriz es y siga siendo simétrica. Podríamos escribirla desde cero, pero eso implicaría repetir algo de código muy similar al que ya hemos escrito.

El sistema de prototipos en JavaScript hace posible crear una *nueva* clase, parecida a la clase anterior, pero con nuevas definiciones para algunas de sus propiedades. El prototipo de la nueva clase deriva del antiguo prototipo, pero agrega una nueva definición para, por ejemplo, el método `set`.

En términos de programación orientada a objetos, esto se llama *herencia*. La nueva clase hereda propiedades y comportamientos de la vieja clase.

```
class MatrizSimetrica extends Matriz {
  constructor(tamaño, elemento = (x, y) => undefined) {
    super(tamaño, tamaño, (x, y) => {
      if (x < y) return elemento(y, x);
      else return elemento(x, y);
    });
  }

  set(x, y, valor) {
    super.set(x, y, valor);
    if (x != y) {
      super.set(y, x, valor);
    }
  }
}

let matriz = new MatrizSimetrica(5, (x, y) => `${x},${y}`);
console.log(matriz.get(2, 3));
// → 3,2
```

El uso de la palabra `extends` indica que esta clase no debe estar basada directamente en el prototipo de `Objeto` predeterminado, pero de alguna otra clase. Esta se llama la *superclase*. La clase derivada es la *subclase*.

Para inicializar una instancia de `MatrizSimetrica`, el constructor llama a su constructor de superclase a través de la palabra clave `super`. Esto es necesario porque si este nuevo objeto se comporta (más o menos) como una `Matriz`, va a necesitar las propiedades de instancia que tienen las matrices. En orden para asegurar que la matriz sea simétrica, el constructor ajusta el método `contenido` para intercambiar las coordenadas de los valores por debajo del diagonal.

El método `set` nuevamente usa `super`, pero esta vez no para llamar al constructor, pero para llamar a un método específico del conjunto de metodos de la superclase. Estamos redefiniendo `set` pero queremos usar el comportamiento original. Ya que `this.set` se refiere al *nuevo* método`set`, llamarlo no funcionaria. Dentro de los métodos de clase, `super` proporciona una forma de llamar a los métodos tal y como se definieron en la superclase.

La herencia nos permite construir tipos de datos ligeramente diferentes a partir de tipos de datos existentes con relativamente poco trabajo. Es una parte fundamental de la tradición orientada a objetos, junto con la encapsulación y el polimorfismo. Pero mientras que los últimos dos son considerados como ideas maravillosas en la actualidad, la herencia es más controversial.

Mientras que la encapsulación y el polimorfismo se pueden usar para *separar* piezas de código entre sí, reduciendo el enredo del programa en general, la herencia fundamentalmente vincula las clases, creando *mas* enredo. Al heredar de una clase, generalmente tienes que saber más sobre cómo funciona que cuando simplemente la usas. La herencia puede ser una herramienta útil, y la uso de vez en cuando en mis propios programas, pero no debería ser la primera herramienta que busques, y probablemente no deberías estar buscando oportunidades para construir jerarquías (árboles genealógicos de clases) de clases en una manera activa.

El operador instanceof

Ocasionalmente es útil saber si un objeto fue derivado de una clase específica. Para esto, JavaScript proporciona un operador binario llamado `instanceof` ("instancia de").

```
console.log(
  new MatrizSimetrica(2) instanceof MatrizSimetrica);
// → true
console.log(new MatrizSimetrica(2) instanceof Matriz);
// → true
console.log(new Matriz(2, 2) instanceof MatrizSimetrica);
// → false
console.log([1] instanceof Array);
```

```
// → true
```

El operador verá a través de los tipos heredados, por lo que una `MatrizSimetrica` es una instancia de `Matriz`. El operador también se puede aplicar a constructores estándar como `Array`. Casi todos los objetos son una instancia de `Object`.

Resumen

Entonces los objetos hacen más que solo tener sus propias propiedades. Ellos tienen prototipos, que son otros objetos. Estos actuarán como si tuvieran propiedades que no tienen mientras su prototipo tenga esa propiedad. Los objetos simples tienen `Object.prototype` como su prototipo.

Los constructores, que son funciones cuyos nombres generalmente comienzan con una mayúscula, se pueden usar con el operador `new` para crear nuevos objetos. El prototipo del nuevo objeto será el objeto encontrado en la propiedad `prototype` del constructor. Puedes hacer un buen uso de esto al poner las propiedades que todos los valores de un tipo dado comparten en su prototipo. Hay una notación de `class` que proporciona una manera clara de definir un constructor y su prototipo.

Puedes definir getters y setters para secretamente llamar a métodos cada vez que se acceda a la propiedad de un objeto. Los métodos estáticos son métodos almacenados en el constructor de clase, en lugar de su prototipo.

El operador `instanceof` puede, dado un objeto y un constructor, decir si ese objeto es una instancia de ese constructor.

Una cosa útil que hacer con los objetos es especificar una interfaz para ellos y decirle a todos que se supone que deben hablar con ese objeto solo a través de esa interfaz. El resto de los detalles que componen tu objeto ahora estan *encapsulados*, escondidos detrás de la interfaz.

Más de un tipo puede implementar la misma interfaz. El código escrito para utilizar una interfaz automáticamente sabe cómo trabajar con cualquier cantidad de objetos diferentes que proporcionen la interfaz. Esto se llama *polimorfismo*.

Al implementar múltiples clases que difieran solo en algunos detalles, puede ser útil escribir las nuevas clases como *subclases* de una clase existente, *heredando* parte de su comportamiento.

Ejercicios

Un tipo vector

Escribe una clase `Vec` que represente un vector en un espacio de dos dimensiones. Toma los parámetros (numericos) `x` y `y`, que debería guardar como propiedades del mismo nombre.

Dale al prototipo de `Vector` dos métodos, `mas` y `menos`, los cuales toman otro vector como parámetro y retornan un nuevo vector que tiene la suma o diferencia de los valores x y y de los dos vectores (`this` y el parámetro).

Agrega una propiedad getter llamada `longitud` al prototipo que calcule la longitud del vector—es decir, la distancia del punto (x, y) desde el origen (0, 0).

Conjuntos

El entorno de JavaScript estándar proporciona otra estructura de datos llamada `Set` ("Conjunto"). Al igual que una instancia de `Map`, un conjunto contiene una colección de valores. Pero a diferencia de `Map`, este no asocia valores con otros—este solo rastrea qué valores son parte del conjunto. Un valor solo puede ser parte de un conjunto una vez—agregarlo de nuevo no tiene ningún efecto.

Escribe una clase llamada `Conjunto`. Como `Set`, debe tener los métodos `add` ("añadir"), `delete` ("eliminar"), y `has` ("tiene"). Su constructor crea un conjunto vacío, `añadir` agrega un valor al conjunto (pero solo si no es ya un miembro), `eliminar` elimina su argumento del conjunto (si era un miembro) y `tiene` retorna un valor booleano que indica si su argumento es un miembro del conjunto.

Usa el operador `===`, o algo equivalente como `indexOf`, para determinar si dos valores son iguales.

Proporcionale a la clase un método estático `desde` que tome un objeto iterable como argumento y cree un grupo que contenga todos los valores producidos al iterar sobre el.

Conjuntos Iterables

Haz iterable la clase `Conjunto` del ejercicio anterior. Puedes remitirte a la sección acerca de la interfaz del iterador anteriormente en el capítulo si ya no recuerdas muy bien la forma exacta de la interfaz.

Si usaste un array para representar a los miembros del conjunto, no solo retornes el iterador creado llamando al método `Symbol.iterator` en el array. Eso funcionaría, pero frustra el propósito de este ejercicio.

Está bien si tu iterador se comporta de manera extraña cuando el conjunto es modificado durante la iteración.

Tomando un método prestado

Anteriormente en el capítulo mencioné que el metodo `hasOwnProperty` de un objeto puede usarse como una alternativa más robusta al operador `in` cuando quieras ignorar las propiedades del prototipo. Pero, ¿y si tu mapa necesita incluir la palabra `"hasOwnProperty"`? Ya no podrás llamar a ese método ya que la propiedad del objeto oculta el valor del método.

¿Puedes pensar en una forma de llamar `hasOwnProperty` en un objeto que tiene una propia propiedad con ese nombre?

"[...] la pregunta de si las Maquinas Pueden Pensar [...] es tan relevante como la pregunta de si los Submarinos Pueden Nadar."
—Edsger Dijkstra, The Threats to Computing Science

Chapter 7

Proyecto: Un Robot

En los capítulos de "proyectos", dejaré de golpearte con teoría nueva por un breve momento y en su lugar vamos a trabajar juntos en un programa. La teoría es necesaria para aprender a programar, pero leer y entender programas reales es igual de importante.

Nuestro proyecto en este capítulo es construir un autómata, un pequeño programa que realiza una tarea en un mundo virtual. Nuestro autómata será un robot de entregas por correo que recoge y deja paquetes.

VillaPradera

El pueblo de VillaPradera no es muy grande. Este consiste de 11 lugares con 14 caminos entre ellos. Puede ser describido con este array de caminos:

```
const caminos = [
  "Casa de Alicia-Casa de Bob",        "Casa de Alicia-Cabaña",
  "Casa de Alicia-Oficina de Correos", "Casa de Bob-Ayuntamiento",
  "Casa de Daria-Casa de Ernie",       "Casa de Daria-Ayuntamiento",
  "Casa de Ernie-Casa de Grete",       "Casa de Grete-Granja",
  "Casa de Grete-Tienda",              "Mercado-Granja",
  "Mercado-Oficina de Correos",        "Mercado-Tienda",
  "Mercado-Ayuntamiento",              "Tienda-Ayuntamiento"
];
```

La red de caminos en el pueblo forma un *grafo*. Un grafo es una colección de puntos (lugares en el pueblo) con líneas entre ellos (caminos). Este grafo será el mundo por el que nuestro robot se movera.

El array de strings no es muy fácil de trabajar. En lo que estamos interesados es en los destinos a los que podemos llegar desde un lugar determinado. Vamos a convertir la lista de caminos en una estructura de datos que, para cada lugar, nos diga a donde se pueda llegar desde allí.

```
function construirGrafo(bordes) {
  let grafo = Object.create(null);
  function añadirBorde(desde, hasta) {
    if (grafo[desde] == null) {
      grafo[desde] = [hasta];
    } else {
      grafo[desde].push(hasta);
    }
  }
  for (let [desde, hasta] of bordes.map(c => c.split("-"))) {
    añadirBorde(desde, hasta);
    añadirBorde(hasta, desde);
  }
  return grafo;
}

const grafoCamino = construirGrafo(roads);
```

Dado un conjunto de bordes, `construirGrafo` crea un objeto de mapa que,

para cada nodo, almacena un array de nodos conectados.

Utiliza el método `split` para ir de los strings de caminos, que tienen la forma `"Comienzo-Final"`, a arrays de dos elementos que contienen el inicio y el final como strings separados.

La tarea

Nuestro robot se moverá por el pueblo. Hay paquetes en varios lugares, cada uno dirigido a otro lugar. El robot tomara paquetes cuando los encuentre y los entregara cuando llegue a sus destinos.

El autómata debe decidir, en cada punto, a dónde ir después. Ha finalizado su tarea cuando se han entregado todos los paquetes.

Para poder simular este proceso, debemos definir un mundo virtual que pueda describirlo. Este modelo nos dice dónde está el robot y dónde estan los paquetes. Cuando el robot ha decidido moverse a alguna parte, necesitamos actualizar el modelo para reflejar la nueva situación.

Si estás pensando en términos de programación orientada a objetos, tu primer impulso podría ser comenzar a definir objetos para los diversos elementos en el mundo. Una clase para el robot, una para un paquete, tal vez una para los lugares. Estas podrían tener propiedades que describen su estado actual, como la pila de paquetes en un lugar, que podríamos cambiar al actualizar el mundo.

Esto está mal.

Al menos, usualmente lo esta. El hecho de que algo suena como un objeto no significa automáticamente que debe ser un objeto en tu programa. Escribir por reflejo las clases para cada concepto en tu aplicación tiende a dejarte con una colección de objetos interconectados donde cada uno tiene su propio estado interno y cambiante. Tales programas a menudo son difíciles de entender y, por lo tanto, fáciles de romper.

En lugar de eso, condensemos el estado del pueblo hasta el mínimo conjunto de valores que lo definan. Está la ubicación actual del robot y la colección de paquetes no entregados, cada uno de los cuales tiene una ubicación actual y una dirección de destino. Eso es todo.

Y mientras estamos en ello, hagámoslo de manera que no *cambiemos* este estado cuándo se mueva el robot, sino calcular un *nuevo* estado para la situación después del movimiento.

```
class EstadoPueblo {
  constructor(lugar, paquetes) {
    this.lugar = lugar;
    this.paquetes = paquetes;
```

```
  }

  mover(destino) {
    if (!grafoCamino[this.lugar].includes(destino)) {
      return this;
    } else {
      let paquetes = this.paquetes.map(p => {
        if (p.lugar != this.lugar) return p;
        return {lugar: destino, direccion: p.direccion};
      }).filter(p => p.lugar != p.direccion);
      return new EstadoPueblo(destino, paquetes);
    }
  }
}
```

En el método `mover` es donde ocurre la acción. Este primero verifica si hay un camino que va del lugar actual al destino, y si no, retorna el estado anterior, ya que este no es un movimiento válido.

Luego crea un nuevo estado con el destino como el nuevo lugar del robot. Pero también necesita crear un nuevo conjunto de paquetes—los paquetes que el robot esta llevando (que están en el lugar actual del robot) necesitan de moverse tambien al nuevo lugar. Y paquetes que están dirigidos al nuevo lugar donde deben de ser entregados—es decir, deben de eliminarse del conjunto de paquetes no entregados. La llamada a `map` se encarga de mover los paquetes, y la llamada a `filter` hace la entrega.

Los objetos de paquete no se modifican cuando se mueven, sino que se vuelven a crear. El método `movee` nos da un nuevo estado de aldea, pero deja el viejo completamente intacto

```
let primero = new EstadoPueblo(
  "Oficina de Correos",
  [{lugar: "Oficina de Correos", direccion: "Casa de Alicia"}]
);
let siguiente = primero.mover("Casa de Alicia");

console.log(siguiente.lugar);
// → Casa de Alicia
console.log(siguiente.parcels);
// → []
console.log(primero.lugar);
// → Oficina de Correos
```

Mover hace que se entregue el paquete, y esto se refleja en el próximo estado. Pero el estado inicial todavía describe la situación donde el robot está en la oficina de correos y el paquete aun no ha sido entregado.

Datos persistentes

Las estructuras de datos que no cambian se llaman *inmutables* o *persistentes*. Se comportan de manera muy similar a los strings y números en que son quienes son, y se quedan así, en lugar de contener diferentes cosas en diferentes momentos.

En JavaScript, casi todo *puede* ser cambiado, así que trabajar con valores que se supone que sean persistentes requieren cierta restricción. Hay una función llamada `Object.freeze` ("Objeto.congelar") que cambia un objeto de manera que escribir en sus propiedades sea ignorado. Podrías usar eso para asegurarte de que tus objetos no cambien, si quieres ser cuidadoso. La congelación requiere que la computadora haga un trabajo extra e ignorar actualizaciones es probable que confunda a alguien tanto como para que hagan lo incorrecto. Por lo general, prefiero simplemente decirle a la gente que un determinado objeto no debe ser molestado, y espero que lo recuerden.

```
let objeto = Object.freeze({valor: 5});
objeto.valor = 10;
console.log(objeto.valor);
// → 5
```

Por qué me salgo de mi camino para no cambiar objetos cuando el lenguaje obviamente está esperando que lo haga?

Porque me ayuda a entender mis programas. Esto es acerca de manejar la complejidad nuevamente. Cuando los objetos en mi sistema son cosas fijas y estables, puedo considerar las operaciones en ellos de forma aislada—moverse a la casa de Alicia desde un estado de inicio siempre produce el mismo nuevo estado. Cuando los objetos cambian con el tiempo, eso agrega una dimensión completamente nueva de complejidad a este tipo de razonamiento.

Para un sistema pequeño como el que estamos construyendo en este capítulo, podriamos manejar ese poco de complejidad adicional. Pero el límite más importante sobre qué tipo de sistemas podemos construir es cuánto podemos entender. Cualquier cosa que haga que tu código sea más fácil de entender hace que sea posible construir un sistema más ambicioso.

Lamentablemente, aunque entender un sistema basado en estructuras de datos persistentes es más fácil, *diseñar* uno, especialmente cuando tu lenguaje

de programación no ayuda, puede ser un poco más difícil. Buscaremos oportunidades para usar estructuras de datos persistentes en este libro, pero también utilizaremos las modificables.

Simulación

Un robot de entregas mira al mundo y decide en qué dirección que quiere moverse. Como tal, podríamos decir que un robot es una función que toma un objeto `EstadoPueblo` y retorna el nombre de un lugar cercano.

Ya que queremos que los robots sean capaces de recordar cosas, para que puedan hacer y ejecutar planes, también les pasamos su memoria y les permitimos retornar una nueva memoria. Por lo tanto, lo que retorna un robot es un objeto que contiene tanto la dirección en la que quiere moverse como un valor de memoria que se le sera regresado la próxima vez que se llame.

```
function correrRobot(estado, robot, memoria) {
  for (let turno = 0;; turno++) {
    if (estado.paquetes.length == 0) {
      console.log(`Listo en ${turno} turnos`);
      break;
    }
    let accion = robot(estado, memoria);
    estado = estado.mover(accion.direccion);
    memoria = accion.memoria;
    console.log(`Moverse a ${accion.direccion}`);
  }
}
```

Considera lo que un robot tiene que hacer para "resolver" un estado dado. Debe recoger todos los paquetes visitando cada ubicación que tenga un paquete, y entregarlos visitando cada lugar al que se dirige un paquete, pero solo después de recoger el paquete.

Cuál es la estrategia más estúpida que podría funcionar? El robot podría simplemente caminar hacia una dirección aleatoria en cada vuelta. Eso significa, con gran probabilidad, que eventualmente se encontrara con todos los paquetes, y luego también en algún momento llegara a todos los lugares donde estos deben ser entregados.

Aqui esta como se podria ver eso:

```
function eleccionAleatoria(array) {
  let eleccion = Math.floor(Math.random() * array.length);
```

```
  return array[eleccion];
}

function robotAleatorio(estado) {
  return {direccion: eleccionAleatoria(grafoCamino[estado.lugar])};
}
```

Recuerda que `Math.random ()` retorna un número entre cero y uno, pero siempre debajo de uno. Multiplicar dicho número por la longitud de un array y luego aplicarle `Math.floor` nos da un índice aleatorio para el array.

Como este robot no necesita recordar nada, ignora su segundo argumento (recuerda que puedes llamar a las funciones en JavaScript con argumentos adicionales sin efectos negativos) y omite la propiedad `memoria` en su objeto retornado.

Para poner en funcionamiento este sofisticado robot, primero necesitaremos una forma de crear un nuevo estado con algunos paquetes. Un método estático (escrito aquí al agregar directamente una propiedad al constructor) es un buen lugar para poner esa funcionalidad.

```
EstadoPueblo.aleatorio = function(numeroDePaquetes = 5) {
  let paquetes = [];
  for (let i = 0; i < numeroDePaquetes; i++) {
    let direccion = eleccionAleatoria(Object.keys(grafoCamino));
    let lugar;
    do {
      lugar = eleccionAleatoria(Object.keys(grafoCamino));
    } while (lugar == direccion);
    paquetes.push({lugar, direccion});
  }
  return new EstadoPueblo("Oficina de Correos", paquetes);
};
```

No queremos paquetes que sean enviados desde el mismo lugar al que están dirigidos. Por esta razón, el bucle `do` sigue seleccionando nuevos lugares cuando obtenga uno que sea igual a la dirección.

Comencemos un mundo virtual.

```
correrRobot(EstadoPueblo.aleatorio(), robotAleatorio);
// → Moverse a Mercado
// → Moverse a Ayuntamiento
// →…
// → Listo en 63 turnos
```

Le toma muchas vueltas al robot para entregar los paquetes, porque este no está planeando muy bien. Nos ocuparemos de eso pronto.

La ruta del camión de correos

Deberíamos poder hacer algo mucho mejor que el robot aleatorio. Una mejora fácil sería tomar una pista de la forma en que como funciona la entrega de correos en el mundo real. Si encontramos una ruta que pasa por todos los lugares en el pueblo, el robot podría ejecutar esa ruta dos veces, y en ese punto esta garantizado que ha entregado todos los paquetes. Aquí hay una de esas rutas (comenzando desde la Oficina de Correos).

```
const rutaCorreo = [
  "Casa de Alicia", "Cabaña", "Casa de Alicia", "Casa de Bob",
  "Ayuntamiento", "Casa de Daria", "Casa de Ernie",
  "GCasa de Grete", "Tienda", "Casa de Grete", "Granja",
  "Mercado", "Oficina de Correos"
];
```

Para implementar el robot que siga la ruta, necesitaremos hacer uso de la memoria del robot. El robot mantiene el resto de su ruta en su memoria y deja caer el primer elemento en cada vuelta.

```
function robotRuta(estado, memoria) {
  if (memoria.length == 0) {
    memoria = rutaCorreo;
  }
  return {direction: memoria[0], memoria: memoria.slice(1)};
}
```

Este robot ya es mucho más rápido. Tomará un máximo de 26 turnos (dos veces la ruta de 13 pasos), pero generalmente seran menos.

Búsqueda de rutas

Aún así, realmente no llamaría seguir ciegamente una ruta fija comportamiento inteligente. El robot podría funcionar más eficientemente si ajustara su comportamiento al trabajo real que necesita hacerse.

Para hacer eso, tiene que ser capaz de avanzar deliberadamente hacia un determinado paquete, o hacia la ubicación donde se debe entregar un paquete. Al hacer eso, incluso cuando el objetivo esté a más de un movimiento de distancia, requiere algún tipo de función de búsqueda de ruta.

El problema de encontrar una ruta a través de un grafo es un típico *problema de búsqueda*. Podemos decir si una solución dada (una ruta) es una solución válida, pero no podemos calcular directamente la solución de la misma manera que podríamos para 2 + 2. En cambio, tenemos que seguir creando soluciones potenciales hasta que encontremos una que funcione.

El número de rutas posibles a través de un grafo es infinito. Pero cuando buscamos una ruta de A a B, solo estamos interesados en aquellas que comienzan en A. Tampoco nos importan las rutas que visitan el mismo lugar dos veces, definitivamente esa no es la ruta más eficiente en cualquier sitio. Entonces eso reduce la cantidad de rutas que el buscador de rutas tiene que considerar.

De hecho, estamos más interesados en la ruta *mas corta*. Entonces queremos asegurarnos de mirar las rutas cortas antes de mirar las más largas. Un buen enfoque sería "crecer" las rutas desde el punto de partida, explorando cada lugar accesible que aún no ha sido visitado, hasta que una ruta alcanze la meta. De esa forma, solo exploraremos las rutas que son potencialmente interesantes, y encontremos la ruta más corta (o una de las rutas más cortas, si hay más de una) a la meta.

Aquí hay una función que hace esto:

```
function encontrarRuta(grafo, desde, hasta) {
  let trabajo = [{donde: desde, ruta: []}];
  for (let i = 0; i < trabajo.length; i++) {
    let {donde, ruta} = trabajo[i];
    for (let lugar of grafo[donde]) {
      if (lugar == hasta) return ruta.concat(lugar);
      if (!trabajo.some(w => w.donde == lugar)) {
        trabajo.push({donde: lugar, ruta: ruta.concat(lugar)});
      }
    }
  }
}
```

La exploración tiene que hacerse en el orden correcto—los lugares que fueron alcanzados primero deben ser explorados primero. No podemos explorar de inmediato un lugar apenas lo alcanzamos, porque eso significaría que los lugares alcanzados *desde allí* también se explorarían de inmediato, y así sucesivamente, incluso aunque puedan haber otros caminos más cortos que aún no han sido

explorados.

Por lo tanto, la función mantiene una *lista de trabajo*. Esta es un array de lugares que deberían explorarse a continuación, junto con la ruta que nos llevó ahí. Esta comienza solo con la posición de inicio y una ruta vacía.

La búsqueda luego opera tomando el siguiente elemento en la lista y explorando eso, lo que significa que todos los caminos que van desde ese lugar son mirados. Si uno de ellos es el objetivo, una ruta final puede ser retornada. De lo contrario, si no hemos visto este lugar antes, un nuevo elemento se agrega a la lista. Si lo hemos visto antes, ya que estamos buscando primero rutas cortas, hemos encontrado una ruta más larga a ese lugar o una precisamente tan larga como la existente, y no necesitamos explorarla.

Puedes imaginar esto visualmente como una red de rutas conocidas que se arrastran desde el lugar de inicio, creciendo uniformemente hacia todos los lados (pero nunca enredándose de vuelta a si misma). Tan pronto como el primer hilo llegue a la ubicación objetivo, ese hilo se remonta al comienzo, dándonos asi nuestra ruta.

Nuestro código no maneja la situación donde no hay más elementos de trabajo en la lista de trabajo, porque sabemos que nuestro gráfico está *conectado*, lo que significa que se puede llegar a todos los lugares desde todos los otros lugares. Siempre podremos encontrar una ruta entre dos puntos, y la búsqueda no puede fallar.

```
function robotOrientadoAMetas({lugar, paquetes}, ruta) {
  if (ruta.length == 0) {
    let paquete = paquetes[0];
    if (paquete.lugar != lugar) {
      ruta = encontrarRuta(grafoCamino, lugar, paquete.lugar);
    } else {
      ruta = encontrarRuta(grafoCamino, lugar, paquete.direccion);
    }
  }
  return {direccion: ruta[0], memoria: ruta.slice(1)};
}
```

Este robot usa su valor de memoria como una lista de instrucciones para moverse, como el robot que sigue la ruta. Siempre que esa lista esté vacía, este tiene que descubrir qué hacer a continuación. Toma el primer paquete no entregado en el conjunto y, si ese paquete no se ha recogido aún, traza una ruta hacia el. Si el paquete *ha* sido recogido, todavía debe ser entregado, por lo que el robot crea una ruta hacia la dirección de entrega en su lugar.

Este robot generalmente termina la tarea de entregar 5 paquetes en 16 turnos

aproximadamente. Un poco mejor que `robotRuta`, pero definitivamente no es óptimo.

Ejercicios

Midiendo un robot

Es difícil comparar objetivamente robots simplemente dejándolos resolver algunos escenarios. Tal vez un robot acaba de conseguir tareas más fáciles, o el tipo de tareas en las que es bueno, mientras que el otro no.

Escribe una función `compararRobots` que toma dos robots (y su memoria de inicio). Debe generar 100 tareas y dejar que cada uno de los robots resuelvan cada una de estas tareas. Cuando terminen, debería generar el promedio de pasos que cada robot tomó por tarea.

En favor de lo que es justo, asegúrate de la misma tarea a ambos robots, en lugar de generar diferentes tareas por robot.

Eficiencia del robot

Puedes escribir un robot que termine la tarea de entrega más rápido que `robotOrientadoAMetas`? Si observas el comportamiento de ese robot, qué obviamente cosas estúpidas este hace? Cómo podrían mejorarse?

Si resolviste el ejercicio anterior, es posible que desees utilizar tu función `compararRobots` para verificar si has mejorado al robot.

Conjunto persistente

La mayoría de las estructuras de datos proporcionadas en un entorno de JavaScript estándar no son muy adecuadas para usos persistentes. Los arrays tienen los métodos `slice` y `concat`, que nos permiten fácilmente crear nuevos arrays sin dañar al anterior. Pero `Set`, por ejemplo, no tiene métodos para crear un nuevo conjunto con un elemento agregado o eliminado.

Escribe una nueva clase `ConjuntoP`, similar a la clase `Conjunto` del Capitulo 6, que almacena un conjunto de valores. Como `Grupo`, tiene métodos `añadir`, `eliminar`, y `tiene`.

Su método `añadir`, sin embargo, debería retornar una *nueva* instancia de `ConjuntoP` con el miembro dado agregado, y dejar la instancia anterior sin cambios. Del mismo modo, `eliminar` crea una nueva instancia sin un miembro dado.

La clase debería funcionar para valores de cualquier tipo, no solo strings. Esta *no* tiene que ser eficiente cuando se usa con grandes cantidades de valores.

El constructor no deberia ser parte de la interfaz de la clase (aunque definitivamente querrás usarlo internamente). En cambio, allí hay una instancia vacía, `ConjuntoP.vacio`, que se puede usar como un valor de inicio.

Por qué solo necesitas un valor `ConjuntoP.vacio`, en lugar de tener una función que crea un nuevo mapa vacío cada vez?

> *"Arreglar errores es dos veces mas difícil que escribir el código en primer lugar. Por lo tanto, si escribes código de la manera más inteligente posible, eres, por definición, no lo suficientemente inteligente como para depurarlo."*
>
> —Brian Kernighan and P.J. Plauger, The Elements of Programming Style

Chapter 8

Bugs y Errores

Los defectos en los programas de computadora usualmente se llaman *bugs* (o "insectos"). Este nombre hace que los programadores se sientan bien al imaginarlos como pequeñas cosas que solo sucede se arrastran hacia nuestro trabajo. En la realidad, por supuesto, nosotros mismos los ponemos allí.

Si un programa es un pensamiento cristalizado, puedes categorizar en grandes rasgos a los bugs en aquellos causados al confundir los pensamientos, y los causados por cometer errores al convertir un pensamiento en código. El primer tipo es generalmente más difícil de diagnosticar y corregir que el último.

Lenguaje

Muchos errores podrían ser señalados automáticamente por la computadora, si esta supiera lo suficiente sobre lo que estamos tratando de hacer. Pero aquí la soltura de JavaScript es un obstáculo. Su concepto de vinculaciones y propiedades es lo suficientemente vago que rara vez atrapará errores ortograficos antes de ejecutar el programa. E incluso entonces, te permite hacer algunas cosas claramente sin sentido, como calcular `true * "mono"`.

Hay algunas cosas de las que JavaScript se queja. Escribir un programa que no siga la gramática del lenguaje inmediatamente hara que la computadora se queje. Otras cosas, como llamar a algo que no sea una función o buscar una propiedad en un valor indefinido, causará un error que sera reportado cuando el programa intente realizar la acción.

Pero a menudo, tu cálculo sin sentido simplemente producirá `NaN` (no es un número) o un valor indefinido. Y el programa continuara felizmente, convencido de que está haciendo algo significativo. El error solo se manifestara más tarde, después de que el valor falso haya viajado a traves de varias funciones. Puede no desencadenar un error en absoluto, pero en silencio causara que la salida del programa sea incorrecta. Encontrar la fuente de tales problemas puede ser algo difícil.

El proceso de encontrar errores—bugs—en los programas se llama *depuración*.

Modo estricto

JavaScript se puede hacer un *poco* más estricto al habilitar el *modo estricto*. Esto se hace al poner el string `"use strict"` ("usar estricto") en la parte superior de un archivo o cuerpo de función. Aquí hay un ejemplo:

```
function puedesDetectarElProblema() {
  "use strict";
  for (contador = 0; contador < 10; contador++) {
    console.log("Feliz feliz");
  }
}

puedesDetectarElProblema();
// → ReferenceError: contador is not defined
```

Normalmente, cuando te olvidas de poner `let` delante de tu vinculación, como con `contador` en el ejemplo, JavaScript silenciosamente crea una vinculación global y utiliza eso. En el modo estricto, se reportara un error en su lugar. Esto es muy útil. Sin embargo, debe tenerse en cuenta que esto no funciona cuando la vinculación en cuestión ya existe como una vinculación global. En ese caso, el ciclo aún sobrescribirá silenciosamente el valor de la vinculación.

Otro cambio en el modo estricto es que la vinculación `this` contiene el valor `undefined` en funciones que no se llamen como métodos. Cuando se hace una llamada fuera del modo estricto, `this` se refiere al objeto del alcance global, que es un objeto cuyas propiedades son vinculaciones globales. Entonces, si llamas accidentalmente a un método o constructor incorrectamente en el modo estricto, JavaScript producirá un error tan pronto trate de leer algo de `this`, en lugar de escribirlo felizmente al alcance global.

Por ejemplo, considera el siguiente código, que llama una función constructora sin la palabra clave `new` de modo que su `this` *no* hara referencia a un objeto recién construido:

```
function Persona(nombre) { this.nombre = nombre; }
let ferdinand = Persona("Ferdinand"); // oops
console.log(nombre);
// → Ferdinand
```

Así que la llamada fraudulenta a `Persona` tuvo éxito pero retorno un valor indefinido y creó la vinculación `nombre` global. En el modo estricto, el resultado es diferente.

```
"use strict";
function Persona(nombre) { this.nombre = nombre; }
let ferdinand = Persona("Ferdinand"); // olvide new
// → TypeError: Cannot set property 'nombre' of undefined
```

Se nos dice inmediatamente que algo está mal. Esto es útil.

Afortunadamente, los constructores creados con la notación `class` siempre se quejan si se llaman sin `new`, lo que hace que esto sea menos un problema incluso en el modo no-estricto.

El modo estricto hace algunas cosas más. No permite darle a una función múltiples parámetros con el mismo nombre y elimina ciertas características problemáticas del lenguaje por completo (como la declaración `with` ("con"), la cual esta tan mal, que no se discute en este libro).

En resumen, poner `"use strict"` en la parte superior de tu programa rara vez duele y puede ayudarte a detectar un problema.

Tipos

Algunos lenguajes quieren saber los tipos de todas tus vinculaciones y expresiones incluso antes de ejecutar un programa. Estos te dirán de una vez cuando uses un tipo de una manera inconsistente. JavaScript solo considera a los tipos cuando ejecuta el programa, e incluso a menudo intentara convertir implícitamente los valores al tipo que espera, por lo que no es de mucha ayuda

Aún así, los tipos proporcionan un marco útil para hablar acerca de los programas. Muchos errores provienen de estar confundido acerca del tipo de valor que entra o sale de una función. Si tienes esa información anotada, es menos probable que te confundas.

Podrías agregar un comentario como arriba de la función `robotOrientadoAMetas` del último capítulo, para describir su tipo.

```
// (EstadoMundo, Array) → {direccion: string, memoria: Array}
function robotOrientadoAMetas(estado, memoria) {
  // ...
}
```

Hay varias convenciones diferentes para anotar programas de JavaScript con tipos.

Una cosa acerca de los tipos es que necesitan introducir su propia complejidad para poder describir suficiente código como para poder ser útil. Cual crees que sería el tipo de la función `eleccionAleatoria` que retorna un elemento aleatorio de un array? Deberías introducir un *tipo variable*, T, que puede representar cualquier tipo, para que puedas darle a `eleccionAleatoria` un tipo como `([T])→T` (función de un array de Ts a a T).

Cuando se conocen los tipos de un programa, es posible que la computadora haga un *chequeo* por ti, señalando los errores antes de que el programa sea ejecutado. Hay varios dialectos de JavaScript que agregan tipos al lenguaje y y los verifica. El más popular se llama TypeScript. Si estás interesado en agregarle más rigor a tus programas, te recomiendo que lo pruebes.

En este libro, continuaremos usando código en JavaScript crudo, peligroso y sin tipos.

Probando

Si el lenguaje no va a hacer mucho para ayudarnos a encontrar errores, tendremos que encontrarlos de la manera difícil: ejecutando el programa y viendo si hace lo correcto.

Hacer esto a mano, una y otra vez, es una muy mala idea. No solo es es molesto, también tiende a ser ineficaz, ya que lleva demasiado tiempo probar exhaustivamente todo cada vez que haces un cambio en tu programa.

Las computadoras son buenas para las tareas repetitivas, y las pruebas son las tareas repetitivas ideales. Las pruebas automatizadas es el proceso de escribir un programa que prueba otro programa. Escribir pruebas consiste en algo más de trabajo que probar manualmente, pero una vez que lo haz hecho, ganas un tipo de superpoder: solo te tomara unos segundos verificar que tu programa todavía se comporta correctamente en todas las situaciones para las que escribiste tu prueba. Cuando rompas algo, lo notarás inmediatamente, en lugar aleatoriomente encontrarte con el problema en algún momento posterior.

Las pruebas usualmente toman la forma de pequeños programas etiquetados que verifican algún aspecto de tu código. Por ejemplo, un conjunto de pruebas para el método (estándar, probablemente ya probado por otra persona) `toUpperCase` podría verse así:

```
function probar(etiqueta, cuerpo) {
  if (!cuerpo()) console.log(`Fallo: ${etiqueta}`);
}
```

```
probar("convertir texto Latino a mayúscula", () => {
  return "hola".toUpperCase() == "HOLA";
});
probar("convertir texto Griego a mayúsculas", () => {
  return "Χαίρετε".toUpperCase() == "ΧΑΊΡΕΤΕ";
});
probar("no convierte caracteres sin mayúsculas", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
});
```

Escribir pruebas de esta manera tiende a producir código bastante repetitivo e incómodo. Afortunadamente, existen piezas de software que te ayudan a construir y ejecutar colecciones de pruebas (*suites de prueba*) al proporcionar un lenguaje (en forma de funciones y métodos) adecuado para expresar pruebas y obtener información informativa cuando una prueba falla. Estos generalmente se llaman *corredores de pruebas*.

Algunos programas son más fáciles de probar que otros programas. Por lo general, con cuantos más objetos externos interactúe el código, más difícil es establecer el contexto en el cual probarlo. El estilo de programación mostrado en el capítulo anterior, que usa valores persistentes auto-contenidos en lugar de cambiar objetos, tiende a ser fácil de probar.

Depuración

Una vez que notes que hay algo mal con tu programa porque se comporta mal o produce errores, el siguiente paso es descubir *cual* es el problema.

A veces es obvio. El mensaje de error apuntará a una línea específica de tu programa, y si miras la descripción del error y esa línea de código, a menudo puedes ver el problema.

Pero no siempre. A veces, la línea que provocó el problema es simplemente el primer lugar en donde un valor extraño producido en otro lugar es usado de una manera inválida. Si has estado resolviendo los ejercicios en capítulos anteriores, probablemente ya habrás experimentado tales situaciones.

El siguiente programa de ejemplo intenta convertir un número entero a un string en una base dada (decimal, binario, etc.) al repetidamente seleccionar el último dígito y luego dividiendo el número para deshacerse de este dígito. Pero la extraña salida que produce sugiere que tiene un error.

```
function numeroAString(n, base = 10) {
```

```
  let resultado = "", signo = "";
  if (n < 0) {
    signo = "-";
    n = -n;
  }
  do {
    resultado = String(n % base) + resultado;
    n /= base;
  } while (n > 0);
  return signo + resultado;
}
console.log(numeroAString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e…-3181.3
```

Incluso si ya ves el problema, finge por un momento que no lo has hecho. Sabemos que nuestro programa no funciona bien, y queremos encontrar por qué.

Aquí es donde debes resistir el impulso de comenzar a hacer cambios aleatorios en el código para ver si eso lo mejora. En cambio, *piensa*. Analiza lo que está sucediendo y piensa en una teoría de por qué podría ser sucediendo. Luego, haz observaciones adicionales para probar esta teoría—o si aún no tienes una teoría, haz observaciones adicionales para ayudarte a que se te ocurra una.

Poner algunas llamadas estratégicas a `console.log` en el programa es una buena forma de obtener información adicional sobre lo que está haciendo el programa. En en este caso, queremos que `n` tome los valores `13`, `1` y luego `0`. Vamos a escribir su valor al comienzo del ciclo.

```
13
1.3
0.13
0.013…

1.5e-323
```

Exacto. Dividir 13 entre 10 no produce un número entero. En lugar de `n /= base`, lo que realmente queremos es `n = Math.floor(n / base)` para que el número sea correctamente "desplazado" hacia la derecha.

Una alternativa al uso de `console.log` para echarle un vistazo al comportamiento del programa es usar las capacidades del *depurador* de tu navegador. Los navegadores vienen con la capacidad de establecer un *punto de interrupción* en una línea específico de tu código. Cuando la ejecución del programa

alcanza una línea con un punto de interrupción, este entra en pausa, y puedes inspeccionar los valores de las vinculaciones en ese punto. No entraré en detalles, ya que los depuradores difieren de navegador en navegador, pero mira las herramientas de desarrollador en tu navegador o busca en la Web para obtener más información.

Otra forma de establecer un punto de interrupción es incluir una declaración `debugger` (que consiste simplemente de esa palabra clave) en tu programa. Si las herramientas de desarrollador en tu navegador están activas, el programa pausará cada vez que llegue a tal declaración.

Propagación de errores

Desafortunadamente, no todos los problemas pueden ser prevenidos por el programador. Si tu programa se comunica con el mundo exterior de alguna manera, es posible obtener una entrada malformada, sobrecargarse con el trabajo, o la red falle en la ejecución.

Si solo estás programando para ti mismo, puedes permitirte ignorar tales problemas hasta que estos ocurran. Pero si construyes algo que va a ser utilizado por cualquier otra persona, generalmente quieres que el programa haga algo mejor que solo estrellarse. A veces lo correcto es tomar la mala entrada en zancada y continuar corriendo. En otros casos, es mejor informar al usuario lo que salió mal y luego darse por vencido. Pero en cualquier situación, el programa tiene que hacer algo activamente en respuesta al problema.

Supongamos que tienes una función `pedirEntero` que le pide al usuario un número entero y lo retorna. Qué deberías retornar si la entrada por parte del usuario es “naranja”?

Una opción es hacer que retorne un valor especial. Opciones comunes para tales valores son `null`, `undefined`, o -1.

```
function pedirEntero(pregunta) {
  let resultado = Number(prompt(pregunta));
  if (Number.isNaN(resultado)) return null;
  else return resultado;
}

console.log(pedirEntero("Cuantos arboles ves?"));
```

Ahora cualquier código que llame a `pedirEntero` debe verificar si un número real fue leído y, si eso falla, de alguna manera debe recuperarse—tal vez preguntando nuevamente o usando un valor predeterminado. O podría de nuevo

retornar un valor especial a *su* llamada para indicar que no pudo hacer lo que se pidió.

En muchas situaciones, principalmente cuando los errores son comunes y la persona que llama debe tenerlos explícitamente en cuenta, retornar un valor especial es una buena forma de indicar un error. Sin embargo, esto tiene sus desventajas. Primero, qué pasa si la función puede retornar cada tipo de valor posible? En tal función, tendrás que hacer algo como envolver el resultado en un objeto para poder distinguir el éxito del fracaso.

```
function ultimoElemento(array) {
  if (array.length == 0) {
    return {fallo: true};
  } else {
    return {elemento: array[array.length - 1]};
  }
}
```

El segundo problema con retornar valores especiales es que puede conducir a código muy incómodo. Si un fragmento de código llama a `pedirEntero` 10 veces, tiene que comprobar 10 veces si `null` fue retornado. Y si su respuesta a encontrar `null` es simplemente retornar `null` en sí mismo, los llamadores de esa función a su vez tendrán que verificarlo, y así sucesivamente.

Excepciones

Cuando una función no puede continuar normalmente, lo que nos *gustaría* hacer es simplemente detener lo que estamos haciendo e inmediatamente saltar a un lugar que sepa cómo manejar el problema. Esto es lo que el *manejo de excepciones* hace.

Las excepciones son un mecanismo que hace posible que el código que se encuentre con un problema *produzca* (o *lance*) una excepción. Una excepción puede ser cualquier valor. Producir una se asemeja a un retorno súper-cargado de una función: salta no solo de la función actual sino también fuera de sus llamadores, todo el camino hasta la primera llamada que comenzó la ejecución actual. Esto se llama *desenrollando la pila*. Puede que recuerdes que la pila de llamadas de función fue mencionada en el Capítulo 3. Una excepción se aleja de esta pila, descartando todos los contextos de llamadas que encuentra.

Si las excepciones siempre se acercaran al final de la pila, estas no serían de mucha utilidad. Simplemente proporcionarían una nueva forma de explotar tu programa. Su poder reside en el hecho de que puedes establecer

"obstáculos" a lo largo de la pila para *capturar* la excepción, cuando esta esta se dirige hacia abajo. Una vez que hayas capturado una excepción, puedes hacer algo con ella para abordar el problema y luego continuar ejecutando el programa.

Aquí hay un ejemplo:

```
function pedirDireccion(pregunta) {
  let resultado = prompt(pregunta);
  if (resultado.toLowerCase() == "izquierda") return "I";
  if (resultado.toLowerCase() == "derecha") return "D";
  throw new Error("Dirección invalida: " + resultado);
}

function mirar() {
  if (pedirDireccion("Hacia que dirección quieres ir?") == "I") {
    return "una casa";
  } else {
    return "dos osos furiosos";
  }
}

try {
  console.log("Tu ves", mirar());
} catch (error) {
  console.log("Algo incorrecto sucedio: " + error);
}
```

La palabra clave `throw` ("producir") se usa para generar una excepción. La captura de una se hace al envolver un fragmento de código en un bloque `try` ("intentar"), seguido de la palabra clave `catch` ("atrapar"). Cuando el código en el bloque `try` cause una excepción para ser producida, se evalúa el bloque `catch`, con el nombre en paréntesis vinculado al valor de la excepción. Después de que el bloque `catch` finaliza, o si el bloque `try` finaliza sin problemas, el programa procede debajo de toda la declaración `try/catch`.

En este caso, usamos el constructor `Error` para crear nuestro valor de excepción. Este es un constructor (estándar) de JavaScript que crea un objeto con una propiedad `message` ("mensaje"). En la mayoría de los entornos de JavaScript, las instancias de este constructor también recopilan información sobre la pila de llamadas que existía cuando se creó la excepción, algo llamado *seguimiento de la pila*. Esta información se almacena en la propiedad `stack` ("pila") y puede ser útil al intentar depurar un problema: esta nos dice la función donde ocurrió el problema y qué funciones realizaron la llamada fallida.

Ten en cuenta que la función `mirar` ignora por completo la posibilidad de que `pedirDireccion` podría salir mal. Esta es la gran ventaja de las excepciones: el código de manejo de errores es necesario solamente en el punto donde el error ocurre y en el punto donde se maneja. Las funciones en el medio puede olvidarse de todo.

Bueno, casi...

Limpiando después de excepciones

El efecto de una excepción es otro tipo de flujo de control. Cada acción que podría causar una excepción, que es prácticamente cualquier llamada de función y acceso a propiedades, puede causar al control dejar tu codigo repentinamente.

Eso significa que cuando el código tiene varios efectos secundarios, incluso si parece que el flujo de control "regular" siempre sucederá, una excepción puede evitar que algunos de ellos sucedan.

Aquí hay un código bancario realmente malo.

```
const cuentas = {
  a: 100,
  b: 0,
  c: 20
};

function obtenerCuenta() {
  let nombreCuenta = prompt("Ingrese el nombre de la cuenta");
  if (!cuentas.hasOwnProperty(nombreCuenta)) {
    throw new Error(`La cuenta "${nombreCuenta}" no existe`);
  }
  return nombreCuenta;
}

function transferir(desde, cantidad) {
  if (cuentas[desde] < cantidad) return;
  cuentas[desde] -= cantidad;
  cuentas[obtenerCuenta()] += cantidad;
}
```

La función `transferir` transfiere una suma de dinero desde una determinada cuenta a otra, pidiendo el nombre de la otra cuenta en el proceso. Si se le da un nombre de cuenta no válido, `obtenerCuenta` arroja una excepción.

Pero `transferir` *primero* remueve el dinero de la cuenta, y *luego* llama a `obtenerCuenta` antes de añadirlo a la otra cuenta. Si esto es interrumpido por

una excepción en ese momento, solo hará que el dinero desaparezca.

Ese código podría haber sido escrito de una manera un poco más inteligente, por ejemplo al llamar `obtenerCuenta` antes de que se comience a mover el dinero. Pero a menudo problemas como este ocurren de maneras más sutiles. Incluso funciones que no parece que lanzarán una excepción podría hacerlo en circunstancias excepcionales o cuando contienen un error de programador.

Una forma de abordar esto es usar menos efectos secundarios. De nuevo, un estilo de programación que calcula nuevos valores en lugar de cambiar los datos existentes ayuda. Si un fragmento de código deja de ejecutarse en el medio de crear un nuevo valor, nadie ve el valor a medio terminar, y no hay ningún problema.

Pero eso no siempre es práctico. Entonces, hay otra característica que las declaraciones `try` tienen. Estas pueden ser seguidas por un bloque `finally` ("finalmente") en lugar de o además de un bloque `catch`. Un bloque `finally` dice "no importa lo que pase, ejecuta este código después de intentar ejecutar el código en el bloque `try`."

```
function transferir(desde, cantidad) {
  if (cuentas[desde] < cantidad) return;
  let progreso = 0;
  try {
    cuentas[desde] -= cantidad;
    progreso = 1;
    cuentas[obtenerCuenta()] += cantidad;
    progreso = 2;
  } finally {
    if (progreso == 1) {
      cuentas[desde] += cantidad;
    }
  }
}
```

Esta versión de la función rastrea su progreso, y si, cuando este terminando, se da cuenta de que fue abortada en un punto donde habia creado un estado de programa inconsistente, repara el daño que hizo.

Ten en cuenta que, aunque el código `finally` se ejecuta cuando una excepción deja el bloque `try`, no interfiere con la excepción. Después de que se ejecuta el bloque `finally`, la pila continúa desenrollandose.

Escribir programas que funcionan de manera confiable incluso cuando aparecen excepciones en lugares inesperados es muy difícil. Muchas personas simplemente no se molestan, y porque las excepciones suelen reservarse para cir-

cunstancias excepcionales, el problema puede ocurrir tan raramente que nunca siquiera es notado. Si eso es algo bueno o algo realmente malo depende de cuánto daño hará el software cuando falle.

Captura selectiva

Cuando una excepción llega hasta el final de la pila sin ser capturada, esta es manejada por el entorno. Lo que esto significa difiere entre los entornos. En los navegadores, una descripción del error generalmente sera escrita en la consola de JavaScript (accesible a través de las herramientas de desarrollador del navegador). Node.js, el entorno de JavaScript sin navegador que discutiremos en el Capítulo 20, es más cuidadoso con la corrupción de datos. Aborta todo el proceso cuando ocurre una excepción no manejada.

Para los errores de programador, solo dejar pasar al error es a menudo lo mejor que puedes hacer. Una excepción no manejada es una forma razonable de señalizar un programa roto, y la consola de JavaScript, en los navegadores moderno, te proporcionan cierta información acerca de qué llamdas de función estaban en la pila cuando ocurrió el problema.

Para problemas que se *espera* que sucedan durante el uso rutinario, estrellarse con una excepción no manejada es una estrategia terrible.

Usos inválidos del lenguaje, como hacer referencia a vinculaciones inexistentes, buscar una propiedad en `null`, o llamar a algo que no sea una función, también dará como resultado que se levanten excepciones. Tales excepciones también pueden ser atrapadas.

Cuando se ingresa en un cuerpo `catch`, todo lo que sabemos es que *algo* en nuestro cuerpo `try` provocó una excepción. Pero no sabemos *que*, o *cual* excepción este causó.

JavaScript (en una omisión bastante evidente) no proporciona soporte directo para la captura selectiva de excepciones: o las atrapas todas o no atrapas nada. Esto hace que sea tentador *asumir* que la excepción que obtienes es en la que estabas pensando cuando escribiste el bloque `catch`.

Pero puede que no. Alguna otra suposición podría ser violada, o es posible que hayas introducido un error que está causando una excepción. Aquí está un ejemplo que *intenta* seguir llamando `pedirDireccion` hasta que obtenga una respuesta válida:

```
for (;;) {
  try {
    let direccion = peirDirrecion("Donde?"); // ← error tipografico!
    console.log("Tu elegiste ", direccion);
```

```
    break;
  } catch (e) {
    console.log ("No es una dirección válida. Inténtalo de nuevo");
  }
}
```

El constructo `for (;;)` es una forma de crear intencionalmente un ciclo que no termine por si mismo. Salimos del ciclo solamente una cuando dirección válida sea dada. *Pero* escribimos mal `pedirDireccion`, lo que dará como resultado un error de "variable indefinida". Ya que el bloque `catch` ignora por completo su valor de excepción (`e`), suponiendo que sabe cuál es el problema, trata erróneamente al error de vinculación como indicador de una mala entrada. Esto no solo causa un ciclo infinito, también "entierra" el útil mensaje de error acerca de la vinculación mal escrita.

Como regla general, no incluyas excepciones a menos de que sean con el propósito de "enrutarlas" hacia alguna parte—por ejemplo, a través de la red para decirle a otro sistema que nuestro programa se bloqueó. E incluso entonces, piensa cuidadosamente sobre cómo podrias estar ocultando información.

Por lo tanto, queremos detectar un tipo de excepción *específico*. Podemos hacer esto al revisar el bloque `catch` si la excepción que tenemos es en la que estamos interesados y relanzar de otra manera. Pero como hacemos para reconocer una excepción?

Podríamos comparar su propiedad `message` con el mensaje de error que sucede estamos esperando. Pero esa es una forma inestable de escribir código—estariamos utilizando información destinada al consumo humano (el mensaje) para tomar una decisión programática. Tan pronto como alguien cambie (o traduzca) el mensaje, el código dejaria de funcionar.

En vez de esto, definamos un nuevo tipo de error y usemos `instanceof` para identificarlo.

```
class ErrorDeEntrada extends Error {}

function pedirDireccion(pregunta) {
  let resultado = prompt(pregunta);
  if (resultado.toLowerCase() == "izquierda") return "I";
  if (resultado.toLowerCase() == "derecha") return "D";
  throw new ErrorDeEntrada("Direccion invalida: " + resultado);
}
```

La nueva clase de error extiende `Error`. No define su propio constructor, lo que significa que hereda el constructor `Error`, que espera un mensaje de string como argumento. De hecho, no define nada—la clase está vacía. Los objetos `ErrorDeEntrada` se comportan como objetos `Error`, excepto que tienen una clase diferente por la cual podemos reconocerlos.

Ahora el ciclo puede atraparlos con mas cuidado.

```
for (;;) {
  try {
    let direccion = pedirDireccion("Donde?");
    console.log("Tu eliges ", direccion);
    break;
  } catch (e) {
    if (e instanceof ErrorDeEntrada) {
      console.log ("No es una dirección válida. Inténtalo de nuevo")
          ;
    } else {
      throw e;
    }
  }
}
```

Esto capturará solo las instancias de `error` y dejará que las excepciones no relacionadas pasen a través. Si reintroduce el error tipográfico, el error de la vinculación indefinida será reportado correctamente.

Afirmaciones

Las *afirmaciones* son comprobaciones dentro de un programa que verifican que algo este en la forma en la que se supone que debe estar. Se usan no para manejar situaciones que puedan aparecer en el funcionamiento normal, pero para encontrar errores hechos por el programador.

Si, por ejemplo, `primerElemento` se describe como una función que nunca se debería invocar en arrays vacíos, podríamos escribirla así:

```
function primerElemento(array) {
  if (array.length == 0) {
    throw new Error("primerElemento llamado con []");
  }
  return array[0];
}
```

Ahora, en lugar de silenciosamente retornar `undefined` (que es lo que obtienes cuando lees una propiedad de array que no existe), esto explotará fuertemente tu programa tan pronto como lo uses mal. Esto hace que sea menos probable que tales errores pasen desapercibidos, y sea más fácil encontrar su causa cuando estos ocurran.

No recomiendo tratar de escribir afirmaciones para todos los tipos posibles de entradas erroneas. Eso sería mucho trabajo y llevaría a código muy ruidoso. Querrás reservarlas para errores que son fáciles de hacer (o que te encuentras haciendo constantemente).

Resumen

Los errores y las malas entradas son hechos de la vida. Una parte importante de la programación es encontrar, diagnosticar y corregir errores. Los problemas pueden será más fáciles de notar si tienes un conjunto de pruebas automatizadas o si agregas afirmaciones a tus programas.

Por lo general, los problemas causados por factores fuera del control del programa deberían ser manejados con gracia. A veces, cuando el problema pueda ser manejado localmente, los valores de devolución especiales son una buena forma de rastrearlos. De lo contrario, las excepciones pueden ser preferibles.

Al lanzar una excepción, se desenrolla la pila de llamadas hasta el próximo bloque `try/catch` o hasta el final de la pila. Se le dará el valor de excepción al bloque `catch` que lo atrape, que debería verificar que en realidad es el tipo esperado de excepción y luego hacer algo con eso. Para ayudar a controlar el impredecible flujo de control causado por las excepciones, los bloques `finally` se pueden usar para asegurarte de que una parte del código *siempre* se ejecute cuando un bloque termina.

Ejercicios

Reintentar

Digamos que tienes una función `multiplicacionPrimitiva` que, en el 20 por ciento de los casos, multiplica dos números, y en el otro 80 por ciento, genera una excepción del tipo `FalloUnidadMultiplicadora`. Escribe una función que envuelva esta torpe función y solo siga intentando hasta que una llamada tenga éxito, después de lo cual retorna el resultado.

Asegúrete de solo manejar las excepciones que estás tratando de manejar.

La caja bloqueada

Considera el siguiente objeto (bastante artificial):

```
const caja = {
  bloqueada: true,
  desbloquear() { this.bloqueada = false; },
  bloquear() { this.bloqueada = true;  },
  _contenido: [],
  get contenido() {
    if (this.bloqueada) throw new Error("Bloqueada!");
    return this._contenido;
  }
};
```

Es solo una caja con una cerradura. Hay un array en la caja, pero solo puedes accederlo cuando la caja esté desbloqueada. Acceder directamente a la propiedad privada `_contenido` está prohibido.

Escribe una función llamada `conCajaDesbloqueada` que toma un valor de función como su argumento, desbloquea la caja, ejecuta la función y luego se asegura de que la caja se bloquee nuevamente antes de retornar, independientemente de si la función argumento retorno normalmente o lanzo una excepción.

"Algunas personas, cuando confrontadas con un problema, piensan 'Ya sé, usaré expresiones regulares.' Ahora tienen dos problemas."

—Jamie Zawinski

Chapter 9

Expresiones Regulares

Las herramientas y técnicas de la programación sobreviven y se propagan de una forma caótica y evolutiva. No siempre son los bonitas o las brillantes las que ganan, sino más bien las que funcionan lo suficientemente bien dentro del nicho correcto o que sucede se integran con otra pieza exitosa de tecnología.

En este capítulo, discutiré una de esas herramientas, *expresiones regulares*. Las expresiones regulares son una forma de describir patrones en datos de tipo string. Estas forman un lenguaje pequeño e independiente que es parte de JavaScript y de muchos otros lenguajes y sistemas.

Las expresiones regulares son terriblemente incómodas y extremadamente útiles. Su sintaxis es críptica, y la interfaz de programación que JavaScript proporciona para ellas es torpe. Pero son una poderosa herramienta para inspeccionar y procesar cadenas. Entender apropiadamente a las expresiones regulares te hará un programador más efectivo.

Creando una expresión regular

Una expresión regular es un tipo de objeto. Puede ser construido con el constructor `RegExp` o escrito como un valor literal al envolver un patrón en caracteres de barras diagonales (`/`).

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

Ambos objetos de expresión regular representan el mismo patrón: un carácter *a* seguido por una *b* seguida de una *c*.

Cuando se usa el constructor `RegExp`, el patrón se escribe como un string normal, por lo que las reglas habituales se aplican a las barras invertidas.

La segunda notación, donde el patrón aparece entre caracteres de barras diagonales, trata a las barras invertidas de una forma diferente. Primero, dado que una barra diagonal termina el patrón, tenemos que poner una barra in-

vertida antes de cualquier barra diagonal que queremos sea *parte* del patrón. En adición, las barras invertidas que no sean parte de códigos especiales de caracteres (como `\n`) seran *preservadas*, en lugar de ignoradas, ya que están en strings, y cambian el significado del patrón. Algunos caracteres, como los signos de interrogación pregunta y los signos de adición, tienen significados especiales en las expresiones regulares y deben ir precedidos por una barra inversa si se pretende que representen al caracter en sí mismo.

```
let dieciochoMas = /dieciocho\+/;
```

Probando por coincidencias

Los objetos de expresión regular tienen varios métodos. El más simple es `test` ("probar"). Si le pasas un string, retornar un Booleano diciéndote si el string contiene una coincidencia del patrón en la expresión.

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

Una expresión regular que consista solamente de caracteres no especiales simplemente representara esa secuencia de caracteres. Si *abc* ocurre en cualquier parte del string con la que estamos probando (no solo al comienzo), `test` retornara `true`.

Conjuntos de caracteres

Averiguar si un string contiene *abc* bien podría hacerse con una llamada a `indexOf`. Las expresiones regulares nos permiten expresar patrones más complicados.

Digamos que queremos encontrar cualquier número. En una expresión regular, poner un conjunto de caracteres entre corchetes hace que esa parte de la expresión coincida con cualquiera de los caracteres entre los corchetes.

Ambas expresiones coincidiran con todas los strings que contengan un dígito:

```
console.log(/[0123456789]/.test("en 1992"));
// → true
console.log(/[0-9]/.test("en 1992"));
```

```
// → true
```

Dentro de los corchetes, un guion (`-`) entre dos caracteres puede ser utilizado para indicar un rango de caracteres, donde el orden es determinado por el número Unicode del carácter. Los caracteres 0 a 9 estan uno al lado del otro en este orden (códigos 48 a 57), por lo que `[0-9]` los cubre a todos y coincide con cualquier dígito.

Un numero de caracteres comunes tienen sus propios atajos incorporados. Los dígitos son uno de ellos: `\d` significa lo mismo que `[0-9]`.

`\d` Cualquier caracter dígito
`\w` Un caracter alfanumérico
`\s` Cualquier carácter de espacio en blanco (espacio, tabulación, nueva línea y similar)
`\D` Un caracter que *no* es un dígito
`\W` Un caracter no alfanumérico
`\S` Un caracter que no es un espacio en blanco
`.` Cualquier caracter a excepción de una nueva línea

Por lo que podrías coincidir con un formato de fecha y hora como 30-01-2003 15:20 con la siguiente expresión:

```
let fechaHora = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(fechaHora.test("30-01-2003 15:20"));
// → true
console.log(fechaHora.test("30-jan-2003 15:20"));
// → false
```

Eso se ve completamente horrible, no? La mitad de la expresión son barras invertidas, produciendo un ruido de fondo que hace que sea difícil detectar el patrón real que queremos expresar. Veremos una versión ligeramente mejorada de esta expresión más tarde.

Estos códigos de barra invertida también pueden usarse dentro de corchetes. Por ejemplo, `[\d.]` representa cualquier dígito o un carácter de punto. Pero el punto en sí mismo, entre corchetes, pierde su significado especial. Lo mismo va para otros caracteres especiales, como `+`.

Para *invertir* un conjunto de caracteres, es decir, para expresar que deseas coincidir con cualquier carácter *excepto* con los que están en el conjunto—puedes escribir un carácter de intercalación (`^`) después del corchete de apertura.

```
let noBinario = /[^01]/;
console.log(noBinario.test("1100100010100110"));
// → false
```

```
console.log(noBinario.test("1100100010200110"));
// → true
```

Repitiendo partes de un patrón

Ya sabemos cómo hacer coincidir un solo dígito. Qué pasa si queremos hacer coincidir un número completo—una secuencia de uno o más dígitos?

Cuando pones un signo más (`+`) después de algo en una expresión regular, este indica que el elemento puede repetirse más de una vez. Por lo tanto, `/\d+/` coincide con uno o más caracteres de dígitos.

```
console.log(/'\d+'/.test("'123'"));
// → true
console.log(/'\d+'/.test("''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("''"));
// → true
```

La estrella (`*`) tiene un significado similar pero también permite que el patrón coincida cero veces. Algo con una estrella después de el nunca evitara un patrón de coincidirlo—este solo coincidirá con cero instancias si no puede encontrar ningun texto adecuado para coincidir.

Un signo de interrogación hace que alguna parte de un patrón sea *opcional*, lo que significa que puede ocurrir cero o mas veces. En el siguiente ejemplo, el carácter *h* está permitido, pero el patrón también retorna verdadero cuando esta letra no esta.

```
let reusar = /reh?usar/;
console.log(reusar.test("rehusar"));
// → true
console.log(reusar.test("reusar"));
// → true
```

Para indicar que un patrón deberia ocurrir un número preciso de veces, usa llaves. Por ejemplo, al poner `{4}` después de un elemento, hace que requiera que este ocurra exactamente cuatro veces. También es posible especificar un rango de esta manera: `{2,4}` significa que el elemento debe ocurrir al menos

dos veces y como máximo cuatro veces.

Aquí hay otra versión del patrón fecha y hora que permite días tanto en dígitos individuales como dobles, meses y horas. Es también un poco más fácil de descifrar.

```
let fechaHora = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(fechaHora.test("30-1-2003 8:45"));
// → true
```

También puedes especificar rangos de final abierto al usar llaves omitiendo el número después de la coma. Entonces, `{5,}` significa cinco o más veces.

Agrupando subexpresiones

Para usar un operador como `*` o `+` en más de un elemento a la vez, tienes que usar paréntesis. Una parte de una expresión regular que se encierre entre paréntesis cuenta como un elemento único en cuanto a los operadores que la siguen están preocupados.

```
let caricaturaLlorando = /boo+(hoo+)+/i;
console.log(caricaturaLlorando.test("Boohoooohoohooo"));
// → true
```

El primer y segundo caracter `+` aplican solo a la segunda *o* en *boo* y *hoo*, respectivamente. El tercer `+` se aplica a la totalidad del grupo `(hoo+)`, haciendo coincidir una o más secuencias como esa.

La `i` al final de la expresión en el ejemplo hace que esta expresión regular sea insensible a mayúsculas y minúsculas, lo que permite que coincida con la letra mayúscula *B* en el string que se le da de entrada, asi el patrón en sí mismo este en minúsculas.

Coincidencias y grupos

El método `test` es la forma más simple de hacer coincidir una expresión. Solo te dice si coincide y nada más. Las expresiones regulares también tienen un método `exec` ("ejecutar") que retorna `null` si no se encontró una coincidencia y retorna un objeto con información sobre la coincidencia de lo contrario.

```
let coincidencia = /\d+/.exec("uno dos 100");
console.log(coincidencia);
```

```
// → ["100"]
console.log(coincidencia.index);
// → 8
```

Un objeto retornado por `exec` tiene una propiedad `index` ("indice") que nos dice *donde* en el string comienza la coincidencia exitosa. Aparte de eso, el objeto parece (y de hecho es) un array de strings, cuyo primer elemento es el string que coincidio—en el ejemplo anterior, esta es la secuencia de dígitos que estábamos buscando.

Los valores de tipo string tienen un método `match` que se comporta de manera similar.

```
console.log("uno dos 100".match(/\d+/));
// → ["100"]
```

Cuando la expresión regular contenga subexpresiones agrupadas con paréntesis, el texto que coincida con esos grupos también aparecerá en el array. La coincidencia completa es siempre el primer elemento. El siguiente elemento es la parte que coincidio con el primer grupo (el que abre paréntesis primero en la expresión), luego el segundo grupo, y asi sucesivamente.

```
let textoCitado = /'([^']*)'/;
console.log(textoCitado.exec("ella dijo 'hola'"));
// → ["'hola'", "hola"]
```

Cuando un grupo no termina siendo emparejado en absoluto (por ejemplo, cuando es seguido de un signo de interrogación), su posición en el array de salida sera `undefined`. Del mismo modo, cuando un grupo coincida multiples veces, solo la ultima coincidencia termina en el array.

```
console.log(/mal(isimo)?/.exec("mal"));
// → ["mal", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

Los grupos pueden ser útiles para extraer partes de un string. Si no solo queremos verificar si un string contiene una fecha pero también extraerla y construir un objeto que la represente, podemos envolver paréntesis alrededor de los patrones de dígitos y tomar directamente la fecha del resultado de `exec`.

Pero primero, un breve desvío, en el que discutiremos la forma incorporada

de representar valores de fecha y hora en JavaScript.

La clase Date (``Fecha'')

JavaScript tiene una clase estándar para representar fechas—o mejor dicho, puntos en el tiempo. Se llama `Date`. Si simplemente creas un objeto fecha usando `new`, obtienes la fecha y hora actual.

```
console.log(new Date());
// → Mon Nov 13 2017 16:19:11 GMT+0100 (CET)
```

También puedes crear un objeto para un tiempo específico.

```
console.log(new Date(2009, 11, 9));
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

JavaScript usa una convención en donde los números de los meses comienzan en cero (por lo que Diciembre es 11), sin embargo, los números de los días comienzan en uno. Esto es confuso y tonto. Ten cuidado.

Los últimos cuatro argumentos (horas, minutos, segundos y milisegundos) son opcionales y se toman como cero cuando no se dan.

Las marcas de tiempo se almacenan como la cantidad de milisegundos desde el inicio de 1970, en la zona horaria UTC. Esto sigue una convención establecida por el "Tiempo Unix", el cual se inventó en ese momento. Puedes usar números negativos para los tiempos anteriores a 1970. Usar el método `getTime` ("obtenerTiempo") en un objeto fecha retorna este número. Es bastante grande, como te puedes imaginar.

```
console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)
```

Si le das al constructor `Date` un único argumento, ese argumento sera tratado como un conteo de milisegundos. Puedes obtener el recuento de milisegundos actual creando un nuevo objeto `Date` y llamando `getTime` en él o llamando a la función `Date.now`.

Los objetos de fecha proporcionan métodos como `getFullYear` ("obtenerAño-

Completo"), `getMonth` ("obtenerMes"), `getDate` ("obtenerFecha"), `getHours` ("obtenerHoras"), `getMinutes` ("obtenerMinutos"), y `getSeconds` ("obtenerSegundos") para extraer sus componentes. Además de `getFullYear`, también existe `getYear` ("obtenerAño"), que te da como resultado un valor de año de dos dígitos bastante inútil (como `93` o `14`).

Al poner paréntesis alrededor de las partes de la expresión en las que estamos interesados, ahora podemos crear un objeto de fecha a partir de un string.

```
function obtenerFecha(string) {
  let [_, dia, mes, año] =
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);
  return new Date(año, mes - 1, dia);
}
console.log(obtenerFecha("30-1-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

La vinculación `_` (guion bajo) es ignorada, y solo se usa para omitir el elemento de coincidencia completa en el array retornado por `exec`.

Palabra y límites de string

Desafortunadamente, `obtenerFecha` felizmente también extraerá la absurda fecha 00-1-3000 del string `"100-1-30000"`. Una coincidencia puede suceder en cualquier lugar del string, por lo que en este caso, esta simplemente comenzará en el segundo carácter y terminara en el penúltimo carácter.

Si queremos hacer cumplir que la coincidencia deba abarcar el string completamente, puedes agregar los marcadores `^` y `$`. El signo de intercalación (`"^"`) coincide con el inicio del string de entrada, mientras que el signo de dólar coincide con el final. Entonces, `/^\d+$/` coincide con un string compuesto por uno o más dígitos, `/^!/` coincide con cualquier string que comience con un signo de exclamación, y `/x^/` no coincide con ningun string (no puede haber una x antes del inicio del string).

Si, por el otro lado, solo queremos asegurarnos de que la fecha comience y termina en un límite de palabras, podemos usar el marcador `\b`. Un límite de palabra puede ser el inicio o el final del string o cualquier punto en el string que tenga un carácter de palabra (como en `\w`) en un lado y un carácter de no-palabra en el otro.

```
console.log(/cat/.test("concatenar"));
// → true
console.log(/\bcat\b/.test("concatenar"));
```

```
// → false
```

Ten en cuenta que un marcador de límite no coincide con un carácter real. Solo hace cumplir que la expresión regular coincida solo cuando una cierta condición se mantenga en el lugar donde aparece en el patrón.

Patrones de elección

Digamos que queremos saber si una parte del texto contiene no solo un número pero un número seguido de una de las palabras *cerdo*, *vaca*, o *pollo*, o cualquiera de sus formas plurales.

Podríamos escribir tres expresiones regulares y probarlas a su vez, pero hay una manera más agradable. El carácter de tubería (`|`) denota una elección entre el patrón a su izquierda y el patrón a su derecha. Entonces puedo decir esto:

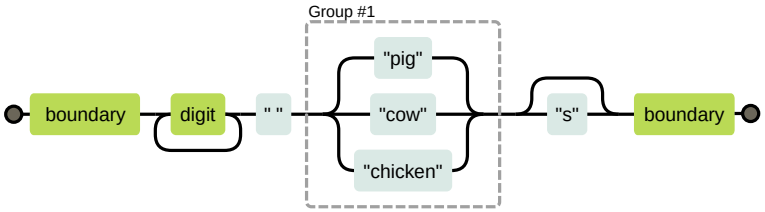
```
let conteoAnimales = /\b\d+ (cerdo|vaca|pollo)s?\b/;
console.log(conteoAnimales.test("15 cerdo"));
// → true
console.log(conteoAnimales.test("15 cerdopollos"));
// → false
```

Los paréntesis se pueden usar para limitar la parte del patrón a la que aplica el operador de tuberia, y puedes poner varios de estos operadores unos a los lados de los otros para expresar una elección entre más de dos alternativas.

Las mecánicas del emparejamiento

Conceptualmente, cuando usas `exec` o `test` el motor de la expresión regular busca una coincidencia en tu string al tratar de hacer coincidir la expresión primero desde el comienzo del string, luego desde el segundo caracter, y así sucesivamente hasta que encuentra una coincidencia o llega al final del string. Retornara la primera coincidencia que se puede encontrar o fallara en encontrar cualquier coincidencia.

Para realmente hacer la coincidencia, el motor trata una expresión regular algo así como un diagrama de flujo. Este es el diagrama para la expresión de ganado en el ejemplo anterior:



Nuestra expresión coincide si podemos encontrar un camino desde el lado izquierdo del diagrama al lado derecho. Mantenemos una posición actual en el string, y cada vez que nos movemos a través de una caja, verificaremos que la parte del string después de nuestra posición actual coincida con esa caja.

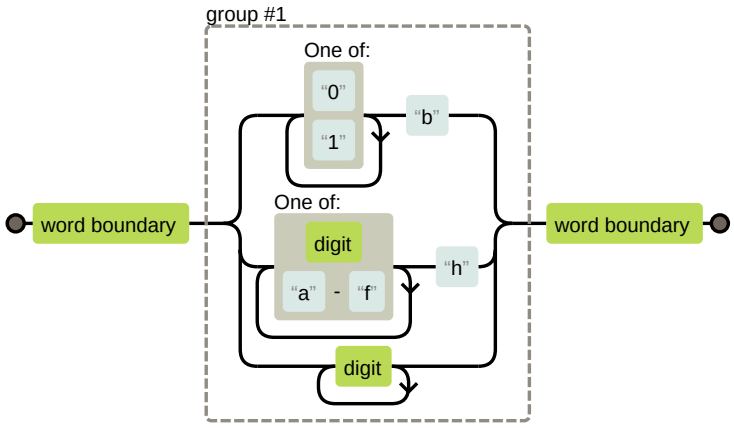
Entonces, si tratamos de coincidir `"los 3 cerdos"` desde la posición 4, nuestro progreso a través del diagrama de flujo se vería así:

- En la posición 4, hay un límite de palabra, por lo que podemos pasar la primera caja.
- Aún en la posición 4, encontramos un dígito, por lo que también podemos pasar la segunda caja.
- En la posición 5, una ruta regresa a antes de la segunda caja (dígito), mientras que la otro se mueve hacia adelante a través de la caja que contiene un caracter de espacio simple. Hay un espacio aquí, no un dígito, asi que debemos tomar el segundo camino.
- Ahora estamos en la posición 6 (el comienzo de “cerdos”) y en el camino de tres vías en el diagrama. No vemos “vaca” o “pollo” aquí, pero vemos “cerdo”, entonces tomamos esa rama.
- En la posición 9, después de la rama de tres vías, un camino se salta la caja *s* y va directamente al límite de la palabra final, mientras que la otra ruta coincide con una *s*. Aquí hay un carácter *s*, no una palabra límite, por lo que pasamos por la caja *s*.
- Estamos en la posición 10 (al final del string) y solo podemos hacer coincidir una palabra límite. El final de un string cuenta como un límite de palabra, así que pasamos por la última caja y hemos emparejado con éxito este string.

Retrocediendo

La expresión regular `/\b([01]+b|[\da-f]+h|\d+)\b/` coincide con un número binario seguido de una *b*, un número hexadecimal (es decir, en base 16, con las

letras *a* a *f* representando los dígitos 10 a 15) seguido de una *h*, o un número decimal regular sin caracter de sufijo. Este es el diagrama correspondiente:



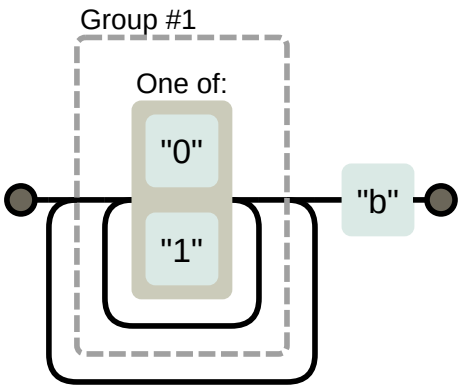
Al hacer coincidir esta expresión, a menudo sucederá que la rama superior (binaria) sea ingresada aunque la entrada en realidad no contenga un número binario. Al hacer coincidir el string `"103"`, por ejemplo, queda claro solo en el 3 que estamos en la rama equivocada. El string *si* coincide con la expresión, pero no con la rama en la que nos encontramos actualmente.

Entonces el "emparejador" *retrocede*. Al ingresar a una rama, este recuerda su posición actual (en este caso, al comienzo del string, justo después del primer cuadro de límite en el diagrama) para que pueda retroceder e intentar otra rama si la actual no funciona. Para el string `"103"`, después de encontrar los 3 caracteres, comenzará a probar la rama para números hexadecimales, que falla nuevamente porque no hay *h* después del número. Por lo tanto, intenta con la rama de número decimal. Esta encaja, y se informa de una coincidencia después de todo.

El emparejador se detiene tan pronto como encuentra una coincidencia completa. Esto significa que si múltiples ramas podrían coincidir con un string, solo la primera (ordenado por donde las ramas aparecen en la expresión regular) es usada.

El retroceso también ocurre para repetición de operadores como + y `*`. Si hace coincidir `/^.*x/` contra `"abcxe"`, la parte `.*` intentará primero consumir todo el string. El motor entonces se dará cuenta de que necesita una *x* para que coincida con el patrón. Como no hay *x* al pasar el final del string, el operador de estrella intenta hacer coincidir un caracter menos. Pero el emparejador tampoco encuentra una *x* después de `abcx`, por lo que retrocede nuevamente, haciendo coincidir el operador de estrella con `abc`. *Ahora* encuentra una *x* donde lo necesita e informa de una coincidencia exitosa de las posiciones 0 a 4.

Es posible escribir expresiones regulares que harán un *monton* de retrocesos. Este problema ocurre cuando un patrón puede coincidir con una pieza de entrada en muchas maneras diferentes. Por ejemplo, si nos confundimos mientras escribimos una expresión regular de números binarios, podríamos accidentalmente escribir algo como `/([01]+)+b/`.



Si intentas hacer coincidir eso con algunas largas series de ceros y unos sin un caracter *b* al final, el emparejador primero pasara por el ciclo interior hasta que se quede sin dígitos. Entonces nota que no hay *b*, asi que retrocede una posición, atraviesa el ciclo externo una vez, y se da por vencido otra vez, tratando de retroceder fuera del ciclo interno una vez más. Continuará probando todas las rutas posibles a través de estos dos bucles. Esto significa que la cantidad de trabajo se *duplica* con cada caracter. Incluso para unas pocas docenas de caracters, la coincidencia resultante tomará prácticamente para siempre.

El método replace

Los valores de string tienen un método `replace` ("reemplazar") que se puede usar para reemplazar parte del string con otro string.

```
console.log("papa".replace("p", "m"));
// → mapa
```

El primer argumento también puede ser una expresión regular, en cuyo caso ña primera coincidencia de la expresión regular es reemplazada. Cuando una opción `g` (para *global*) se agrega a la expresión regular, *todas* las coincidencias en el string será reemplazadas, no solo la primera.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
```

```
// → Barabadar
```

Hubiera sido sensato si la elección entre reemplazar una coincidencia o todas las coincidencias se hiciera a través de un argumento adicional en `replace` o proporcionando un método diferente, `replaceAll` ("reemplazarTodas"). Pero por alguna desafortunada razón, la elección se basa en una propiedad de los expresiones regulares en su lugar.

El verdadero poder de usar expresiones regulares con `replace` viene del hecho de que podemos referirnos a grupos coincidentes en la string de reemplazo. Por ejemplo, supongamos que tenemos una gran string que contenga los nombres de personas, un nombre por línea, en el formato `Apellido, Nombre`. Si deseamos intercambiar estos nombres y eliminar la coma para obtener un formato `Nombre Apellido`, podemos usar el siguiente código:

```
console.log(
  "Liskov, Barbara\nMcCarthy, John\nWadler, Philip"
    .replace(/(\w+), (\w+)/g, "$2 $1"));
// → Barbara Liskov
//   John McCarthy
//   Philip Wadler
```

Los `$1` y `$2` en el string de reemplazo se refieren a los grupos entre paréntesis del patrón. `$1` se reemplaza por el texto que coincide con el primer grupo, `$2` por el segundo, y así sucesivamente, hasta `$9`. Puedes hacer referencia a la coincidencia completa con `$&`.

Es posible pasar una función, en lugar de un string, como segundo argumento para `replace`. Para cada reemplazo, la función será llamada con los grupos coincidentes (así como con la coincidencia completa) como argumentos, y su valor de retorno se insertará en el nuevo string.

Aquí hay un pequeño ejemplo:

```
let s = "la cia y el fbi";
console.log(s.replace(/\b(fbi|cia)\b/g,
            str => str.toUpperCase()));
// → la CIA y el FBI
```

Y aquí hay uno más interesante:

```
let almacen = "1 limon, 2 lechugas, y 101 huevos";
function menosUno(coincidencia, cantidad, unidad) {
  cantidad = Number(cantidad) - 1;
```

```
  if (cantidad == 1) { // solo queda uno, remover la 's'
    unidad = unidad.slice(0, unidad.length - 1);
  } else if (cantidad == 0) {
    cantidad = "sin";
  }
  return cantidad + " " + unidad;
}
console.log(almacen.replace(/(\d+) (\w+)/g, menosUno));
// → sin limon, 1 lechuga, y 100 huevos
```

Esta función toma un string, encuentra todas las ocurrencias de un número seguido de una palabra alfanumérica, y retorna un string en la que cada ocurrencia es decrementada por uno.

El grupo `(\d+)` termina como el argumento `cantidad` para la función, y el grupo `(\w+)` se vincula a `unidad`. La función convierte `cantidad` a un número—lo que siempre funciona, ya que coincidio con `\d+`—y realiza algunos ajustes en caso de que solo quede uno o cero.

Codicia

Es posible usar `replace` para escribir una función que elimine todo los comentarios de un fragmento de código JavaScript. Aquí hay un primer intento:

```
function removerComentarios(codigo) {
  return codigo.replace(/\/\/.*|\/\*[^]*\*\//g, "");
}
console.log(removerComentarios("1 + /* 2 */3"));
// → 1 + 3
console.log(removerComentarios("x = 10;// ten!"));
// → x = 10;
console.log(removerComentarios("1 /* a */+/* b */ 1"));
// → 1  1
```

La parte anterior al operador *o* coincide con dos caracteres de barra inclinada seguido de cualquier número de caracteres que no sean nuevas lineas. La parte para los comentarios de líneas múltiples es más complicado. Usamos `[^]` (cualquier caracter que no está en el conjunto de caracteres vacíos) como una forma de unir cualquier caracter. No podemos simplemente usar un punto aquí porque los comentarios de bloque pueden continuar en una nueva línea, y el carácter del período no coincide con caracteres de nuevas lineas.

Pero la salida de la última línea parece haber salido mal. Por qué?

La parte `[^]*` de la expresión, como describí en la sección retroceder, primero coincidirá tanto como sea posible. Si eso causa un falo en la siguiente parte del patrón, el emparejador retrocede un caracter e intenta nuevamente desde allí. En el ejemplo, el emparejador primero intenta emparejar el resto del string y luego se mueve hacia atrás desde allí. Este encontrará una ocurrencia de `*/` después de retroceder cuatro caracteres y emparejar eso. Esto no es lo que queríamos, la intención era hacer coincidir un solo comentario, no ir hasta el final del código y encontrar el final del último comentario de bloque.

Debido a este comportamiento, decimos que los operadores de repetición (`+`, `*`, `?` y `{}`) son _ codiciosos, lo que significa que coinciden con tanto como pueden y retroceden desde allí. Si colocas un signo de interrogación después de ellos (`+?`, `*?`, `??`, `{}?`), se vuelven no-codiciosos y comienzan a hacer coincidir lo menos posible, haciendo coincidir más solo cuando el patrón restante no se ajuste a la coincidencia más pequeña.

Y eso es exactamente lo que queremos en este caso. Al hacer que la estrella coincida con el tramo más pequeño de caracteres que nos lleve a un `*/`, consumimos un comentario de bloque y nada más.

```
function removerComentarios(codigo) {
  return codigo.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(removerComentarios("1 /* a */+/* b */ 1"));
// → 1 + 1
```

Una gran cantidad de errores en los programas de expresiones regulares se pueden rastrear a intencionalmente usar un operador codicioso, donde uno que no sea codicioso trabajaria mejor. Al usar un operador de repetición, considera la variante no-codiciosa primero.

Creando objetos RegExp dinámicamente

Hay casos en los que quizás no sepas el patrón exacto necesario para coincidir cuando estes escribiendo tu código. Imagina que quieres buscar el nombre del usuario en un texto y encerrarlo en caracteres de subrayado para que se destaque. Como solo sabrás el nombrar una vez que el programa se está ejecutando realmente, no puedes usar la notación basada en barras.

Pero puedes construir un string y usar el constructor `RegExp` en el. Aquí hay un ejemplo:

```
let nombre = "harry";
```

```
let texto = "Harry es un personaje sospechoso.";
let regexp = new RegExp("\\b(" + nombre + ")\\b", "gi");
console.log(texto.replace(regexp, "_$1_"));
// → _Harry_ es un personaje sospechoso.
```

Al crear los marcadores de límite `\b`, tenemos que usar dos barras invertidas porque las estamos escribiendo en un string normal, no en una expresión regular contenida en barras. El segundo argumento para el constructor `RegExp` contiene las opciones para la expresión regular—en este caso, `"gi"` para global e insensible a mayúsculas y minúsculas.

Pero, y si el nombre es `"dea+hl[]rd"` porque nuestro usuario es un nerd adolescente? Eso daría como resultado una expresión regular sin sentido que en realidad no coincidirá con el nombre del usuario.

Para solucionar esto, podemos agregar barras diagonales inversas antes de cualquier caracter que tenga un significado especial.

```
let nombre = "dea+hl[]rd";
let texto = "Este sujeto dea+hl[]rd es super fastidioso.";
let escapados = nombre.replace(/[\\[.+*?(){|^$]/g, "\\$&");
let regexp = new RegExp("\\b" + escapados + "\\b", "gi");
console.log(texto.replace(regexp, "_$&_"));
// → Este sujeto _dea+hl[]rd_ es super fastidioso.
```

El método search

El método `indexOf` en strings no puede invocarse con una expresión regular. Pero hay otro método, `search` (“buscar”), que espera una expresión regular. Al igual que `indexOf`, retorna el primer índice en que se encontró la expresión, o -1 cuando no se encontró.

```
console.log("  palabra".search(/\S/));
// → 2
console.log("    ".search(/\S/));
// → -1
```

Desafortunadamente, no hay forma de indicar que la coincidencia debería comenzar a partir de un desplazamiento dado (como podemos con el segundo argumento para `indexOf`), que a menudo sería útil.

La propiedad lastIndex

De manera similar el método `exec` no proporciona una manera conveniente de comenzar buscando desde una posición dada en el string. Pero proporciona una manera *in*conveniente.

Los objetos de expresión regular tienen propiedades. Una de esas propiedades es `source` ("fuente"), que contiene el string de donde se creó la expresión. Otra propiedad es `lastIndex` ("ultimoIndice"), que controla, en algunas circunstancias limitadas, donde comenzará la siguiente coincidencia.

Esas circunstancias son que la expresión regular debe tener la opción global (`g`) o adhesiva (`y`) habilitada, y la coincidencia debe suceder a través del método `exec`. De nuevo, una solución menos confusa hubiese sido permitir que un argumento adicional fuera pasado a `exec`, pero la confusión es una característica esencial de la interfaz de las expresiones regulares de JavaScript.

```
let patron = /y/g;
patron.lastIndex = 3;
let coincidencia = patron.exec("xyzzy");
console.log(coincidencia.index);
// → 4
console.log(patron.lastIndex);
// → 5
```

Si la coincidencia fue exitosa, la llamada a `exec` actualiza automáticamente a la propiedad `lastIndex` para que apunte después de la coincidencia. Si no se encontraron coincidencias, `lastIndex` vuelve a cero, que es también el valor que tiene un objeto de expresión regular recién construido.

La diferencia entre las opciones globales y las adhesivas es que, cuandoa adhesivo está habilitado, la coincidencia solo tendrá éxito si comienza directamente en `lastIndex`, mientras que con global, buscará una posición donde pueda comenzar una coincidencia.

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let adhesivo = /abc/y;
console.log(adhesivo.exec("xyz abc"));
// → null
```

Cuando se usa un valor de expresión regular compartido para múltiples llamadas a `exec`, estas actualizaciones automáticas a la propiedad `lastIndex`

pueden causar problemas. Tu expresión regular podría estar accidentalmente comenzando en un índice que quedó de una llamada anterior.

```
let digito = /\d/g;
console.log(digito.exec("aqui esta: 1"));
// → ["1"]
console.log(digito.exec("y ahora: 1"));
// → null
```

Otro efecto interesante de la opción global es que cambia la forma en que funciona el método `match` en strings. Cuando se llama con una expresión global, en lugar de retornar un matriz similar al retornado por `exec`,`match` encontrará *todas* las coincidencias del patrón en el string y retornar un array que contiene los strings coincidentes.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

Por lo tanto, ten cuidado con las expresiones regulares globales. Los casos donde son necesarias—llamadas a `replace` y lugares donde deseas explícitamente usar `lastIndex`—son generalmente los únicos lugares donde querras usarlas.

Ciclos sobre coincidencias

Una cosa común que hacer es escanear todas las ocurrencias de un patrón en un string, de una manera que nos de acceso al objeto de coincidencia en el cuerpo del ciclo. Podemos hacer esto usando `lastIndex` y `exec`.

```
let entrada = "Un string con 3 numeros en el... 42 y 88.";
let numero = /\b\d+\b/g;
let coincidencia;
while (coincidencia = numero.exec(entrada)) {
  console.log("Se encontro", coincidencia[0], "en", coincidencia.
      index);
}
// → Se encontro 3 en 14
//   Se encontro 42 en 33
//   Se encontro 88 en 38
```

Esto hace uso del hecho de que el valor de una expresión de asignación (`=`) es el valor asignado. Entonces al usar `coincidencia = numero.exec(entrada)`

como la condición en la declaración `while`, realizamos la coincidencia al inicio de cada iteración, guardamos su resultado en una vinculación, y terminamos de repetir cuando no se encuentran más coincidencias.

Análisis de un archivo INI

Para concluir el capítulo, veremos un problema que requiere de expresiones regulares. Imagina que estamos escribiendo un programa para recolectar automáticamente información sobre nuestros enemigos de el Internet. (No escribiremos ese programa aquí, solo la parte que lee el archivo de configuración. Lo siento.) El archivo de configuración se ve así:

```
motordebusqueda=https://duckduckgo.com/?q=$1
malevolencia=9.7

; los comentarios estan precedidos por un punto y coma...
; cada seccion contiene un enemigo individual

[larry]
nombrecompleto=Larry Doe
tipo=bravucon del preescolar
sitioweb=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]
nombrecompleto=Davaeorn
tipo=hechizero malvado
directoriosalida=/home/marijn/enemies/davaeorn
```

Las reglas exactas para este formato (que es un formato ampliamente utilizado, usualmente llamado un archivo *INI*) son las siguientes:

- Las líneas en blanco y líneas que comienzan con punto y coma se ignoran.
- Las líneas envueltas en `[` y `]` comienzan una nueva sección.
- Líneas que contienen un identificador alfanumérico seguido de un carácter `=` agregan una configuración a la sección actual.
- Cualquier otra cosa no es válida.

Nuestra tarea es convertir un string como este en un objeto cuyas propiedades contengas strings para configuraciones sin sección y sub-objetos para secciones, con esos subobjetos conteniendo la configuración de la sección.

Dado que el formato debe procesarse línea por línea, dividir el archivo en líneas separadas es un buen comienzo. Usamos `string.split("\n")` para hacer esto en el Capítulo 4. Algunos sistemas operativos, sin embargo, usan no solo un carácter de nueva línea para separar lineas sino un carácter de retorno de carro seguido de una nueva línea (`"\r\n"`). Dado que el método `split` también permite una expresión regular como su argumento, podemos usar una expresión regular como `/\r?\n/` para dividir el string de una manera que permita tanto `"\n"` como `"\r\n"` entre líneas.

```
function analizarINI(string) {
  // Comenzar con un objeto para mantener los campos de nivel
     superior
  let resultado = {};
  let seccion = resultado;
  string.split(/\r?\n/).forEach(linea => {
    let coincidencia;
    if (coincidencia = linea.match(/^(\w+)=(.*)$/)) {
      seccion[coincidencia[1]] = coincidencia[2];
    } else if (coincidencia = linea.match(/^\[(.*)\]$/)) {
      seccion = resultado[coincidencia[1]] = {};
    } else if (!/^\s*(;.*)?$/.test(linea)) {
      throw new Error("Linea '" + linea + "' no es valida.");
    }
  });
  return resultado;
}

console.log(analizarINI(`
nombre=Vasilis
[direccion]
ciudad=Tessaloniki`));
// → {nombre: "Vasilis", direccion: {ciudad: "Tessaloniki"}}
```

El código pasa por las líneas del archivo y crea un objeto. Las propiedades en la parte superior se almacenan directamente en ese objeto, mientras que las propiedades que se encuentran en las secciones se almacenan en un objeto de sección separado. La vinculación `sección` apunta al objeto para la sección actual.

Hay dos tipos de de líneas significativas—encabezados de seccion o lineas de propiedades. Cuando una línea es una propiedad regular, esta se almacena en la sección actual. Cuando se trata de un encabezado de sección, se crea un nuevo objeto de sección, y `seccion` se configura para apuntar a él.

Nota el uso recurrente de `^` y `$` para asegurarse de que la expresión coincida con toda la línea, no solo con parte de ella. Dejando afuera estos resultados en código que funciona principalmente, pero que se comporta de forma extraña para algunas entradas, lo que puede ser un error difícil de rastrear.

El patrón `if (coincidencia = string.match (...))` es similar al truco de usar una asignación como condición para `while`. A menudo no estas seguro de que tu llamada a `match` tendrá éxito, para que puedas acceder al objeto resultante solo dentro de una declaración `if` que pruebe esto. Para no romper la agradable cadena de las formas `else if`, asignamos el resultado de la coincidencia a una vinculación e inmediatamente usamos esa asignación como la prueba para la declaración `if`.

Si una línea no es un encabezado de sección o una propiedad, la función verifica si es un comentario o una línea vacía usando la expresión `/^\s*(;.*)?$/`. Ves cómo funciona? La parte entre los paréntesis coincidirá con los comentarios, y el `?` asegura que también coincida con líneas que contengan solo espacios en blanco. Cuando una línea no coincida con cualquiera de las formas esperadas, la función arroja una excepción.

Caracteres internacionales

Debido a la simplista implementación inicial de JavaScript y al hecho de que este enfoque simplista fue luego establecido en piedra como comportamiento estándar, las expresiones regulares de JavaScript son bastante tontas acerca de los caracteres que no aparecen en el idioma inglés. Por ejemplo, en cuanto a las expresiones regulares de JavaScript, una "palabra caracter" es solo uno de los 26 caracteres en el alfabeto latino (mayúsculas o minúsculas), dígitos decimales, y, por alguna razón, el carácter de guion bajo. Cosas como *é* o *ß*, que definitivamente son caracteres de palabras, no coincidirán con `\w` (y *si* coincidiran con `\W` mayúscula, la categoría no-palabra).

Por un extraño accidente histórico, `\s` (espacio en blanco) no tiene este problema y coincide con todos los caracteres que el estándar Unicode considera espacios en blanco, incluyendo cosas como el (espacio de no separación) y el Separador de vocales Mongol.

Otro problema es que, de forma predeterminada, las expresiones regulares funcionan en unidades del código, como se discute en el Capítulo 5, no en caracteres reales. Esto significa que los caracteres que estan compustos de dos unidades de código se comportan de manera extraña.

```
console.log(/🍎{3}/.test("🍎🍎🍎"));
// → false
```

```
console.log(/<.>/.test("<🌹>"));
// → false
console.log(/<.>/u.test("<🌹>"));
// → true
```

El problema es que la 🍎 en la primera línea se trata como dos unidades de código, y la parte `{3}` se aplica solo a la segunda. Del mismo modo, el punto coincide con una sola unidad de código, no con las dos que componen al emoji de rosa.

Debe agregar una opción `u` (para Unicode) a tu expresión regular para hacerla tratar a tales caracteres apropiadamente. El comportamiento incorrecto sigue siendo el predeterminado, desafortunadamente, porque cambiarlo podría causar problemas en código ya existente que depende de él.

Aunque esto solo se acaba de estandarizar y aun no es, al momento de escribir este libro, ampliamente compatible con muchs nabegadores, es posible usar `\p` en una expresión regular (que debe tener habilitada la opción Unicode) para que coincida con todos los caracteres a los que el estándar Unicode lis asigna una propiedad determinada.

```
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
console.log(/\p{Alphabetic}/u.test("α"));
// → true
console.log(/\p{Alphabetic}/u.test("!"));
// → false
```

Unicode define una cantidad de propiedades útiles, aunque encontrar la que necesitas puede no ser siempre trivial. Puedes usar la notación `\p{Property=Value}` para hacer coincidir con cualquier carácter que tenga el valor dado para esa propiedad. Si el nombre de la propiedad se deja afuera, como en `\p{Name}`, se asume que el nombre es una propiedad binaria como `Alfabético` o una categoría como `Número`.

Resumen

Las expresiones regulares son objetos que representan patrones en strings. Ellas usan su propio lenguaje para expresar estos patrones.

`/abc/`	Una secuencia de caracteres
`/[abc]/`	Cualquier caracter de un conjunto de caracteres
`/[^abc]/`	Cualquier carácter que *no* este en un conjunto de caracteres
`/[0-9]/`	Cualquier caracter en un rango de caracteres
`/x+/`	Una o más ocurrencias del patrón `x`
`/x+?/`	Una o más ocurrencias, no codiciosas
`/x*/`	Cero o más ocurrencias
`/x?/`	Cero o una ocurrencia
`/x{2,4}/`	De dos a cuatro ocurrencias
`/(abc)/`	Un grupo
`/a|b|c/`	Cualquiera de varios patrones
`/\d/`	Cualquier caracter de digito
`/\w/`	Un caracter alfanumérico ("carácter de palabra")
`/\s/`	Cualquier caracter de espacio en blanco
`/./`	Cualquier caracter excepto líneas nuevas
`/\b/`	Un límite de palabra
`/^/`	Inicio de entrada
`/$/`	Fin de la entrada

Una expresión regular tiene un método `test` para probar si una determinada string coincide cn ella. También tiene un método `exec` que, cuando una coincidencia es encontrada, retorna un array que contiene todos los grupos que coincidieron. Tal array tiene una propiedad `index` que indica en dónde comenzó la coincidencia.

Los strings tienen un método `match` para coincidirlas con una expresión regular y un método `search` para buscar por una, retornando solo la posición inicial de la coincidencia. Su método `replace` puede reemplazar coincidencias de un patrón con un string o función de reemplazo.

Las expresiones regulares pueden tener opciones, que se escriben después de la barra que cierra la expresión. La opción `i` hace que la coincidencia no distinga entre mayúsculas y minúsculas. La opción `g` hace que la expresión sea *global*, que, entre otras cosas, hace que el método `replace` reemplace todas las instancias en lugar de solo la primera. La opción `y` la hace adhesivo, lo que significa que hará que no busque con anticipación y omita la parte del string cuando busque una coincidencia. La opción `u` activa el modo Unicode, lo que soluciona varios problemas alrededor del manejo de caracteres que toman dos unidades de código.

Las expresiones regulares son herramientas afiladas con un manejo incómodo. Ellas simplifican algunas tareas enormemente, pero pueden volverse inmanejables rápidamente cuando se aplican a problemas complejos. Parte de saber cómo usarlas es resistiendo el impulso de tratar de calzar cosas que no pueden

ser expresadas limpiamente en ellas.

Ejercicios

Es casi inevitable que, durante el curso de trabajar en estos ejercicios, te sentiras confundido y frustrado por el comportamiento inexplicable de alguna regular expresión. A veces ayuda ingresar tu expresión en una herramienta en línea como *debuggex.com* para ver si su visualización corresponde a lo que pretendías y a experimentar con la forma en que responde a varios strings de entrada.

Golf Regexp

Golf de Codigo es un término usado para el juego de intentar expresar un programa particular con el menor número de caracteres posible. Similarmente, *Golf de Regexp* es la práctica de escribir una expresión regular tan pequeña como sea posible para que coincida con un patrón dado, y *sólo* con ese patrón.

Para cada uno de los siguientes elementos, escribe una expresión regular para probar si alguna de las substrings dadas ocurre en un string. La expresión regular debe coincidir solo con strings que contengan una de las substrings descritas. No te preocupes por los límites de palabras a menos que sean explícitamente mencionados. Cuando tu expresión funcione, ve si puedes hacerla más pequeña.

1. *car* y *cat*
2. *pop* y *prop*
3. *ferret*, *ferry*, y *ferrari*
4. Cualquier palabra que termine *ious*
5. Un carácter de espacio en blanco seguido de un punto, coma, dos puntos o punto y coma
6. Una palabra con mas de seis letras
7. Una palabra sin la letra *e* (o *E*)

Consulta la tabla en el resumen del capítulo para ayudarte. Pruebe cada solución con algunos strings de prueba.

Estilo entre comillas

Imagina que has escrito una historia y has utilizado comillass simples en todas partes para marcar piezas de diálogo. Ahora quieres reemplazar todas las comillas de diálogo con comillas dobles, pero manteniendo las comillas simples usadas en contracciones como *aren't*.

Piensa en un patrón que distinga de estos dos tipos de uso de citas y crea una llamada al método `replace` que haga el reemplazo apropiado.

Números otra vez

Escribe una expresión que coincida solo con el estilo de números en JavaScript. Esta debe admitir un signo opcional menos *o* más delante del número, el punto decimal, y la notación de exponente—`5e-3` o `1E10`— nuevamente con un signo opcional en frente del exponente. También ten en cuenta que no es necesario que hayan dígitos delante o detrás del punto, pero el el número no puede ser solo un punto. Es decir, `.5` y `5.` son numeros válidos de JavaScript, pero solo un punto *no* lo es.

"Escriba código que sea fácil de borrar, no fácil de extender."
—Tef, Programming is Terrible

Chapter 10
Módulos

El programa ideal tiene una estructura cristalina. La forma en que funciona es fácil de explicar, y cada parte juega un papel bien definido.

Un típico programa real crece orgánicamente. Nuevas piezas de funcionalidad se agregan a medida que surgen nuevas necesidades. Estructurar—y preservar la estructura—es trabajo adicional, trabajo que solo valdra la pena en el futuro, la *siguiente* vez que alguien trabaje en el programa. Así que es tentador descuidarlo, y permitir que las partes del programa se vuelvan profundamente enredadas.

Esto causa dos problemas prácticos. En primer lugar, entender tal sistema es difícil. Si todo puede tocar todo lo demás, es difícil ver a cualquier pieza dada de forma aislada. Estas obligado a construir un entendimiento holístico de todo el asunto. En segundo lugar, si quieres usar cualquiera de las funcionalidades de dicho programa en otra situación, reescribirla podria resultar más fácil que tratar de desenredarla de su contexto.

El término "gran bola de barro" se usa a menudo para tales programas grandes, sin estructura. Todo se mantiene pegado, y cuando intentas sacar una pieza, todo se desarma y tus manos se ensucian.

Módulos

Los *módulos* son un intento de evitar estos problemas. Un módulo es una pieza del programa que especifica en qué otras piezas este depende (sus *dependencias*) y qué funcionalidad proporciona para que otros módulos usen (su *interfaz*).

Las interfaces de los módulos tienen mucho en común con las interfaces de objetos, como las vimos en el Capítulo 6. Estas hacen parte del módulo disponible para el mundo exterior y mantienen el resto privado. Al restringir las formas en que los módulos interactúan entre sí, el el sistema se parece más a un juego de LEGOS, donde las piezas interactúan a través de conectores bien definidos, y menos como barro, donde todo se mezcla con todo.

Las relaciones entre los módulos se llaman *dependencias*. Cuando un módulo

necesita una pieza de otro módulo, se dice que depende de ese módulo. Cuando este hecho está claramente especificado en el módulo en sí, puede usarse para descubrir qué otros módulos deben estar presentes para poder ser capaces de usar un módulo dado y cargar dependencias automáticamente.

Para separar módulos de esa manera, cada uno necesita su propio alcance privado.

Simplemente poner todo tu código JavaScript en diferentes archivos no satisface estos requisitos. Los archivos aún comparten el mismo espacio de nombres global. Pueden, intencionalmente o accidentalmente, interferir con las vinculaciones de cada uno. Y la estructura de dependencia sigue sin estar clara. Podemos hacerlo mejor, como veremos más adelante en el capítulo.

Diseñar una estructura de módulo ajustada para un programa puede ser difícil. En la fase en la que todavía estás explorando el problema, intentando cosas diferentes para ver que funciona, es posible que desees no preocuparte demasiado por eso, ya que puede ser una gran distracción. Una vez que tengas algo que se sienta sólido, es un buen momento para dar un paso atrás y organizarlo.

Paquetes

Una de las ventajas de construir un programa a partir de piezas separadas, y ser capaces de ejecutar esas piezas por si mismas, es que tú podrías ser capaz de aplicar la misma pieza en diferentes programas.

Pero cómo se configura esto? Digamos que quiero usar la función `analizarINI` del Capítulo 9 en otro programa. Si está claro de qué depende la función (en este caso, nada), puedo copiar todo el código necesario en mi nuevo proyecto y usarlo. Pero luego, si encuentro un error en ese código, probablemente lo solucione en el programa en el que estoy trabajando en ese momento y me olvido de arreglarlo en el otro programa.

Una vez que comience a duplicar código, rápidamente te encontraras perdiendo tiempo y energía moviendo las copias alrededor y manteniéndolas actualizadas.

Ahí es donde los *paquetes* entran. Un paquete es un pedazo de código que puede ser distribuido (copiado e instalado). Puede contener uno o más módulos, y tiene información acerca de qué otros paquetes depende. Un paquete también suele venir con documentación que explica qué es lo que hace, para que las personas que no lo escribieron todavía puedan hacer uso de el.

Cuando se encuentra un problema en un paquete, o se agrega una nueva característica, el el paquete es actualizado. Ahora los programas que dependen de él (que también pueden ser otros paquetes) pueden actualizar a la nueva

versión.

Trabajar de esta manera requiere infraestructura. Necesitamos un lugar para almacenar y encontrar paquetes, y una forma conveniente de instalar y actualizarlos. En el mundo de JavaScript, esta infraestructura es provista por NPM (*npmjs.org*).

NPM es dos cosas: un servicio en línea donde uno puede descargar (y subir) paquetes, y un programa (incluido con Node.js) que te ayuda a instalar y administrarlos.

Al momento de escribir esto, hay más de medio millón de paquetes diferentes disponibles en NPM. Una gran parte de ellos son basura, debería mencionar, pero casi todos los paquetes útiles, disponibles públicamente, se puede encontrar allí. Por ejemplo, un analizador de archivos INI, similar al uno que construimos en el Capítulo 9, está disponible bajo el nombre de paquete `ini`.

En el Capítulo 20 veremos cómo instalar dichos paquetes de forma local utilizando el programa de línea de comandos `npm`.

Tener paquetes de calidad disponibles para descargar es extremadamente valioso. Significa que a menudo podemos evitar tener que reinventar un programa que cien personas han escrito antes, y obtener una implementación sólida y bien probado con solo presionar algunas teclas.

El software es barato de copiar, por lo que una vez lo haya escrito alguien, distribuirlo a otras personas es un proceso eficiente. Pero escribirlo en el primer lugar, *es* trabajo y responder a las personas que han encontrado problemas en el código, o que quieren proponer nuevas características, es aún más trabajo.

Por defecto, tu posees el copyright del código que escribes, y otras personas solo pueden usarlo con tu permiso. Pero ya que algunas personas son simplemente agradables, y porque la publicación de un buen software puede ayudarte a hacerte un poco famoso entre los programadores, se publican muchos paquetes bajo una licencia que explícitamente permite a otras personas usarlos.

La mayoría del código en NPM esta licenciado de esta manera. Algunas licencias requieren que tu publiques también el código bajo la misma licencia del paquete que estas usando. Otras son menos exigentes, solo requieren que guardes la licencia con el código cuando lo distribuyas. La comunidad de JavaScript principalmente usa ese último tipo de licencia. Al usar paquetes de otras personas, asegúrete de conocer su licencia.

Módulos improvisados

Hasta 2015, el lenguaje JavaScript no tenía un sistema de módulos incorporado. Sin embargo, la gente había estado construyendo sistemas grandes en

JavaScript durante más de una década y ellos *necesitaban* módulos.

Así que diseñaron sus propios sistema de módulos arriba del lenguaje. Puedes usar funciones de JavaScript para crear alcances locales, y objetos para representar las interfaces de los módulos.

Este es un módulo para ir entre los nombres de los días y números (como son retornados por el método `getDay` de `Date`). Su interfaz consiste en `diaDeLaSemana.nombre` y `diaDeLaSemana.numero`, y oculta su vinculación local `nombres` dentro del alcance de una expresión de función que se invoca inmediatamente.

```
const diaDeLaSemana = function() {
  const nombres = ["Domingo", "Lunes", "Martes", "Miercoles",
                   "Jueves", "Viernes", "Sabado"];
  return {
    nombre(numero) { return nombres[numero]; },
    numero(nombre) { return nombres.indexOf(nombre); }
  };
}();

console.log(diaDeLaSemana.nombre(diaDeLaSemana.numero("Domingo")));
// → Domingo
```

Este estilo de módulos proporciona aislamiento, hasta cierto punto, pero no declara dependencias. En cambio, simplemente pone su interfaz en el alcance global y espera que sus dependencias, si hay alguna, hagan lo mismo. Durante mucho tiempo, este fue el enfoque principal utilizado en la programación web, pero ahora está mayormente obsoleto.

Si queremos hacer que las relaciones de dependencia sean parte del código, tendremos que tomar el control de las dependencias que deben ser cargadas. Hacer eso requiere que seamos capaces de ejecutar strings como código. JavaScript puede hacer esto.

Evaluando datos como código

Hay varias maneras de tomar datos (un string de código) y ejecutarlos como una parte del programa actual.

La forma más obvia es usar el operador especial `eval`, que ejecuta un string en el alcance *actual*. Esto usualmente es una mala idea porque rompe algunas de las propiedades que normalmente tienen los alcances, tal como fácilmente predecir a qué vinculación se refiere un nombre dado.

```
const x = 1;
```

```
function evaluarYRetornarX(codigo) {
  eval(codigo);
  return x;
}

console.log(evaluarYRetornarX("var x = 2"));
// → 2
```

Una forma menos aterradora de interpretar datos como código es usar el constructor `Function`. Este toma dos argumentos: un string que contiene una lista de nombres de argumentos separados por comas y un string que contiene el cuerpo de la función.

```
let masUno = Function("n", "return n + 1;");
console.log(masUno(4));
// → 5
```

Esto es precisamente lo que necesitamos para un sistema de módulos. Podemos envolver el código del módulo en una función y usar el alcance de esa función como el alcance del módulo.

CommonJS

El enfoque más utilizado para incluir módulos en JavaScript es llamado *módulos CommonJS*. Node.js lo usa, y es el sistema utilizado por la mayoría de los paquetes en NPM.

El concepto principal en los módulos CommonJS es una función llamada `require` ("requerir"). Cuando la llamas con el nombre del módulo de una dependencia, esta se asegura de que el módulo sea cargado y retorna su interfaz.

Debido a que el cargador envuelve el código del módulo en una función, los módulos obtienen automáticamente su propio alcance local. Todo lo que tienen que hacer es llamar a `require` para acceder a sus dependencias, y poner su interfaz en el objeto vinculado a `exports` ("exportaciones").

Este módulo de ejemplo proporciona una función de formateo de fecha. Utiliza dos paquetes de NPM—`ordinal` para convertir números a strings como `"1st"` y `"2nd"`, y `date-names` para obtener los nombres en inglés de los días de la semana y meses. Este exporta una sola función, `formatDate`, que toma un objeto `Date` y un string plantilla.

El string de plantilla puede contener códigos que dirigen el formato, como `YYYY` para todo el año y `Do` para el día ordinal del mes. Podrías darle un string

como "MMMM Do YYYY" para obtener resultados como “November 22nd 2017”.

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

La interfaz de `ordinal` es una función única, mientras que `date-names` exporta un objeto que contiene varias cosas—los dos valores que usamos son arrays de nombres. La desestructuración es muy conveniente cuando creamos vinculaciones para interfaces importadas.

El módulo agrega su función de interfaz a `exports`, de modo que los módulos que dependen de el tengan acceso a el. Podríamos usar el módulo de esta manera:

```
const {formatDate} = require("./format-date");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

Podemos definir `require`, en su forma más mínima, así:

```
require.cache = Object.create(null);

function require(nombre) {
  if (!(nombre in require.cache)) {
    let codigo = leerArchivo(nombre);
    let modulo = {exportaciones: {}};
    require.cache[nombre] = modulo;
    let envolvedor = Function("require, exportaciones, modulo",
       codigo);
    envolvedor(require, modulo.exportaciones, modulo);
  }
  return require.cache[nombre].exportaciones;
```

```
}
```

En este código, `leerArchivo` es una función inventada que lee un archivo y retorna su contenido como un string. El estándar de JavaScript no ofrece tal funcionalidad—pero diferentes entornos de JavaScript, como el navegador y Node.js, proporcionan sus propias formas de acceder a archivos. El ejemplo solo pretende que `leerArchivo` existe.

Para evitar cargar el mismo módulo varias veces, `require` mantiene un (caché) almacenado de módulos que ya han sido cargados. Cuando se llama, primero verifica si el módulo solicitado ya ha sido cargado y, si no, lo carga. Esto implica leer el código del módulo, envolverlo en una función y llamárla.

La interfaz del paquete `ordinal` que vimos antes no es un objeto, sino una función. Una peculiaridad de los módulos CommonJS es que, aunque el sistema de módulos creará un objeto de interfaz vacío para ti (vinculado a `exports`), puedes reemplazarlo con cualquier valor al sobrescribir `module.exports`. Esto lo hacen muchos módulos para exportar un valor único en lugar de un objeto de interfaz.

Al definir `require`, `exportaciones` y `modulo` como parametros para la función de envoltura generada (y pasando los valores apropiados al llamarla), el cargador se asegura de que estas vinculaciones esten disponibles en el alcance del módulo.

La forma en que el string dado a `require` se traduce a un nombre de archivo real o dirección web difiere en diferentes sistemas. Cuando comienza con `"./"` o `"../"`, generalmente se interpreta como relativo al nombre del archivo actual. Entonces `"./format-date"` sería el archivo llamado `format-date.js` en el mismo directorio.

Cuando el nombre no es relativo, Node.js buscará por un paquete instalado con ese nombre. En el código de ejemplo de este capítulo, interpretaremos esos nombres como referencias a paquetes de NPM. Entraremos en más detalles sobre cómo instalar y usar los módulos de NPM en el Capítulo 20.

Ahora, en lugar de escribir nuestro propio analizador de archivos INI, podemos usar uno de NPM:

```
const {parse} = require("ini");

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

Módulos ECMAScript

Los módulos CommonJS funcionan bastante bien y, en combinación con NPM, han permitido que la comunidad de JavaScript comience a compartir código en una gran escala.

Pero siguen siendo un poco de un truco con cinta adhesiva. La notación es ligeramente incomoda—las cosas que agregas a `exports` no están disponibles en el alcance local, por ejemplo. Y ya que `require` es una llamada de función normal tomando cualquier tipo de argumento, no solo un string literal, puede ser difícil de determinar las dependencias de un módulo sin correr su código primero.

Esta es la razón por la cual el estándar de JavaScript introdujo su propio, sistema de módulos diferente a partir de 2015. Por lo general es llamado *módulos ES*, donde *ES* significa ECMAScript. Los principales conceptos de dependencias e interfaces siguen siendo los mismos, pero los detalles difieren. Por un lado, la notación está ahora integrada en el lenguaje. En lugar de llamar a una función para acceder a una dependencia, utilizas una palabra clave `import` ("importar") especial.

```
import ordinal from "ordinal";
import {days, months} from "date-names";

export function formatDate(date, format) { /* ... */ }
```

Similarmente, la palabra clave `export` se usa para exportar cosas. Puede aparecer delante de una función, clase o definición de vinculación (`let`, `const`, o `var`).

La interfaz de un módulo ES no es un valor único, sino un conjunto de vinculaciones con nombres. El módulo anterior vincula `formatDate` a una función. Cuando importas desde otro módulo, importas la *vinculación*, no el valor, lo que significa que un módulo exportado puede cambiar el valor de la vinculación en cualquier momento, y que los módulos que la importen verán su nuevo valor.

Cuando hay una vinculación llamada `default`, esta se trata como el principal valor del módulo exportado. Si importas un módulo como `ordinal` en el ejemplo, sin llaves alrededor del nombre de la vinculación, obtienes su vinculación `default`. Dichos módulos aún pueden exportar otras vinculaciones bajo diferentes nombres ademas de su exportación por `default`.

Para crear una exportación por default, escribe `export default` antes de una expresión, una declaración de función o una declaración de clase.

```
export default ["Invierno", "Primavera", "Verano", "Otoño"];
```

Es posible renombrar la vinculación importada usando la palabra `as` ("como").

```
import {days as nombresDias} from "date-names";

console.log(nombresDias.length);
// → 7
```

Al momento de escribir esto, la comunidad de JavaScript está en proceso de adoptar este estilo de módulos. Pero ha sido un proceso lento. Tomó algunos años, después de que se haya especificado el formato paraq que los navegadores y Node.js comenzaran a soportarlo. Y a pesar de que lo soportan mayormente ahora, este soporte todavía tiene problemas, y la discusión sobre cómo dichos módulos deberían distribuirse a través de NPM todavía está en curso.

Muchos proyectos se escriben usando módulos ES y luego se convierten automáticamente a algún otro formato cuando son publicados. Estamos en período de transición en el que se utilizan dos sistemas de módulos diferentes uno al lado del otro, y es útil poder leer y escribir código en cualquiera de ellos.

Construyendo y empaquetando

De hecho, muchos proyectos de JavaScript ni siquiera están, técnicamente, escritos en JavaScript. Hay extensiones, como el dialecto de comprobación de tipos mencionado en el Capítulo 7, que son ampliamente usados. Las personas también suelen comenzar a usar extensiones planificadas para el lenguaje mucho antes de que estas hayan sido agregadas a las plataformas que realmente corren JavaScript.

Para que esto sea posible, ellos *compilan* su código, traduciéndolo del dialecto de JavaScript que eligieron a JavaScript simple y antiguo—o incluso a una versión anterior de JavaScript, para que navegadores antiguos puedan ejecutarlo.

Incluir un programa modular que consiste de 200 archivos diferentes en una página web produce sus propios problemas. Si buscar un solo archivo sobre la red tarda 50 milisegundos, cargar todo el programa tardaria 10 segundos, o tal vez la mitad si puedes cargar varios archivos simultáneamente. Eso es mucho tiempo perdido. Ya que buscar un solo archivo grande tiende a ser más rápido que buscar muchos archivos pequeños, los programadores web han comenzado a usar herramientas que convierten sus programas (los cuales cuidadosamente

estan divididos en módulos) de nuevo en un único archivo grande antes de publicarlo en la Web. Tales herramientas son llamado *empaquetadores*.

Y podemos ir más allá. Además de la cantidad de archivos, el *tamaño* de los archivos también determina qué tan rápido se pueden transferir a través de la red. Por lo tanto, la comunidad de JavaScript ha inventado *minificadores*. Estas son herramientas que toman un programa de JavaScript y lo hacen más pequeño al eliminar automáticamente los comentarios y espacios en blanco, cambia el nombre de las vinculaciones, y reemplaza piezas de código con código equivalente que ocupa menos espacio.

Por lo tanto, no es raro que el código que encuentres en un paquete de NPM o que se ejecute en una página web haya pasado por *multiples* etapas de transformación: conversión de JavaScript moderno a JavaScript histórico, del formato de módulos ES a CommonJS, empaquetado y minificado. No vamos a entrar en los detalles de estas herramientas en este libro, ya que tienden a ser aburridos y cambian rápidamente. Solo ten en cuenta que el código JavaScript que ejecutas a menudo no es el código tal y como fue escrito.

Diseño de módulos

La estructuración de programas es uno de los aspectos más sutiles de la programación. Cualquier pieza de funcionalidad no trivial se puede modelar de varias maneras.

Un buen diseño de programa es subjetivo—hay ventajas/desventajas involucradas, y cuestiones de gusto. La mejor manera de aprender el valor de una buena estructura de diseño es leer o trabajar en muchos programas y notar lo que funciona y lo qué no. No asumas que un desastroso doloroso es “solo la forma en que las cosas son ". Puedes mejorar la estructura de casi todo al ponerle mas pensamiento.

Un aspecto del diseño de módulos es la facilidad de uso. Si estás diseñando algo que está destinado a ser utilizado por varias personas—o incluso por ti mismo, en tres meses cuando ya no recuerdes los detalles de lo que hiciste—es útil si tu interfaz es simple y predicible.

Eso puede significar seguir convenciones existentes. Un buen ejemplo es el paquete `ini`. Este módulo imita el objeto estándar `JSON` al proporcionar las funciones `parse` y `stringify` (para escribir un archivo INI), y, como `JSON` , convierte entre strings y objetos simples. Entonces la interfaz es pequeña y familiar, y después de haber trabajado con ella una vez, es probable que recuerdes cómo usarla.

Incluso si no hay una función estándar o un paquete ampliamente utilizado

para imitar, puedes mantener tus módulos predecibles mediante el uso de estructuras de datos simples y haciendo una cosa única y enfocada. Muchos de los módulos de análisis de archivos INI en NPM proporcionan una función que lee directamente tal archivo del disco duro y lo analiza, por ejemplo. Esto hace que sea imposible de usar tales módulos en el navegador, donde no tenemos acceso directo al sistema de archivos, y agrega una complejidad que habría sido mejor abordada al *componer* el módulo con alguna función de lectura de archivos.

Lo que apunta a otro aspecto útil del diseño de módulos—la facilidad con la qué algo se puede componer con otro código. Módulos enfocados que que computan valores son aplicables en una gama más amplia de programas que módulos mas grandes que realizan acciones complicadas con efectos secundarios. Un lector de archivos INI que insista en leer el archivo desde el disco es inútil en un escenario donde el contenido del archivo provenga de alguna otra fuente.

Relacionadamente, los objetos con estado son a veces útiles e incluso necesarios, pero si se puede hacer algo con una función, usa una función. Varios de los lectores de archivos INI en NPM proporcionan un estilo de interfaz que requiere que primero debes crear un objeto, luego cargar el archivo en tu objeto, y finalmente usar métodos especializados para obtener los resultados. Este tipo de cosas es común en la tradición orientada a objetos, y es terrible. En lugar de hacer una sola llamada de función y seguir adelante, tienes que realizar el ritual de mover tu objeto a través de diversos estados. Y ya que los datos ahora están envueltos en un objeto de tipo especializado, todo el código que interactúa con él tiene que saber sobre ese tipo, creando interdependencias innecesarias.

A menudo no se puede evitar la definición de nuevas estructuras de datos—solo unas pocas básicas son provistos por el estándar de lenguaje, y muchos tipos de datos tienen que ser más complejos que un array o un mapa. Pero cuando el array es suficiente, usa un array.

Un ejemplo de una estructura de datos un poco más compleja es el grafo de el Capítulo 7. No hay una sola manera obvia de representar un grafo en JavaScript. En ese capítulo, usamos un objeto cuya propiedades contenian arrays de strings—los otros nodos accesibles desde ese nodo.

Hay varios paquetes de busqueda de rutas diferentes en NPM, pero ninguno de ellos usa este formato de grafo. Por lo general, estos permiten que los bordes del grafo tengan un peso, el costo o la distancia asociada a ellos, lo que no es posible en nuestra representación.

Por ejemplo, está el paquete `dijkstrajs`. Un enfoque bien conocido par la busqueda de rutas, bastante similar a nuestra función `encontrarRuta`, se llama el *algoritmo de Dijkstra*, después de Edsger Dijkstra, quien fue el primero que

lo escribió. El sufijo `js` a menudo se agrega a los nombres de los paquetes para indicar el hecho de que están escritos en JavaScript. Este paquete `dijkstrajs` utiliza un formato de grafo similar al nuestro, pero en lugar de arrays, utiliza objetos cuyos valores de propiedad son números—los pesos de los bordes.

Si quisiéramos usar ese paquete, tendríamos que asegurarnos de que nuestro grafo fue almacenado en el formato que este espera.

```
const {find_path} = require("dijkstrajs");

let grafo = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}

console.log(find_path(grafo, "Oficina de Correos", "Cabaña"));
// → ["Oficina de Correos", "Casa de Alice", "Cabaña"]
```

Esto puede ser una barrera para la composición—cuando varios paquetes están usando diferentes estructuras de datos para describir cosas similares, combinarlos es difícil. Por lo tanto, si deseas diseñar para la compibilidad, averigua qué estructura de datos están usando otras personas y, cuando sea posible, sigue su ejemplo.

Resumen

Los módulos proporcionan de estructura a programas más grandes al separar el código en piezas con interfaces y dependencias claras. La interfaz es la parte del módulo que es visible desde otros módulos, y las dependencias son los otros módulos este que utiliza.

Debido a que históricamente JavaScript no proporcionó un sistema de módulos, el sistema CommonJS fue construido encima. Entonces, en algún momento, *consiguio* un sistema incorporado, que ahora coexiste incomodamente con el sistema CommonJS.

Un paquete es una porción de código que se puede distribuir por sí misma. NPM es un repositorio de paquetes de JavaScript. Puedes descargar todo tipo de paquetes útiles (e inútiles) de él.

Ejercicios

Un robot modular

Estas son las vinculaciones que el proyecto del Capítulo 7 crea:

```
caminos
construirGrafo
grafoCamino
EstadoPueblo
correrRobot
eleccionAleatoria
robotAleatorio
rutaCorreo
robotRuta
encontrarRuta
robotOrientadoAMetas
```

Si tuvieras que escribir ese proyecto como un programa modular, qué módulos crearías? Qué módulo dependería de qué otro módulo, y cómo se verían sus interfaces?

Qué piezas es probable que estén disponibles pre-escritas en NPM? Preferirias usar un paquete de NPM o escribirlas tu mismo?

Módulo de Caminos

Escribe un módulo CommonJS, basado en el ejemplo del Capítulo 7, que contenga el array de caminos y exporte la estructura de datos grafo que los representa como `grafoCamino`. Debería depender de un modulo `./grafo`, que exporta una función `construirGrafo` que se usa para construir el grafo. Esta función espera un array de arrays de dos elementos (los puntos de inicio y final de los caminos).

Dependencias circulares

Una dependencia circular es una situación en donde el módulo A depende de B, y B también, directa o indirectamente, depende de A. Muchos sistemas de módulos simplemente prohíbne esto porque cualquiera que sea el orden que elijas para cargar tales módulos, no puedes asegurarse de que las dependencias de cada módulo han sido cargadas antes de que se ejecuten.

Los modulos CommonJS permiten una forma limitada de dependencias cíclicas. Siempre que los módulos no reemplacen a su objeto `exports` predetermi-

nado, y no accedan a la interfaz de las demás hasta que terminen de cargar, las dependencias cíclicas están bien.

La función `require` dada anteriormente en este capítulo es compatible con este tipo de ciclo de dependencias. Puedes ver cómo maneja los ciclos? Qué iría mal cuando un módulo en un ciclo *reemplace* su objeto `exports` por defecto?

"Quién puede esperar tranquilamente mientras el barro se asienta?
Quién puede permanecer en calma hasta el momento de actuar?"

—Laozi, Tao Te Ching

Chapter 11

Programación Asincrónica

La parte central de una computadora, la parte que lleva a cabo los pasos individuales que componen nuestros programas, es llamada *procesador*. Los programas que hemos visto hasta ahora son cosas que mantienen al procesador ocupado hasta que hayan terminado su trabajo. La velocidad a la que algo como un ciclo que manipule números pueda ser ejecutado, depende casi completamente de la velocidad del procesador.

Pero muchos programas interactúan con cosas fuera del procesador. por ejemplo, podrian comunicarse a través de una red de computadoras o solicitar datos del disco duro—lo que es mucho más lento que obtenerlos desde la memoria.

Cuando una cosa como tal este sucediendo, sería una pena dejar que el procesador se mantenga inactivo—podría haber algún otro trabajo que este pueda hacer en el mientras tanto. En parte, esto es manejado por tu sistema operativo, que cambiará el procesador entre múltiples programas en ejecución. Pero eso no ayuda cuando queremos que un *unico* programa pueda hacer progreso mientras este espera una solicitud de red.

Asincronicidad

En un modelo de programación *sincrónico*, las cosas suceden una a la vez. Cuando llamas a una función que realiza una acción de larga duración, solo retorna cuando la acción ha terminado y puede retornar el resultado. Esto detiene tu programa durante el tiempo que tome la acción.

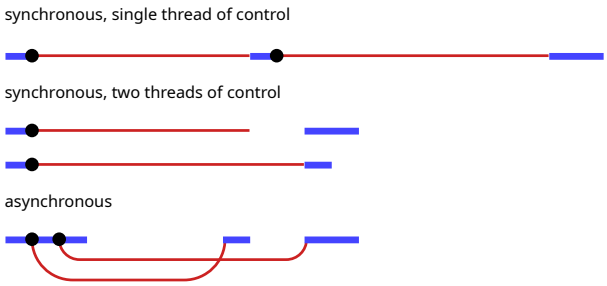
Un modelo *asincrónico* permite que ocurran varias cosas al mismo tiempo. Cuando comienzas una acción, tu programa continúa ejecutándose. Cuando la acción termina, el programa es informado y tiene acceso al resultado (por ejemplo, los datos leídos del disco).

Podemos comparar a la programación síncrona y asincrónica usando un pequeño ejemplo: un programa que obtiene dos recursos de la red y luego combina resultados.

En un entorno síncrono, donde la función de solicitud solo retorna una vez que ha hecho su trabajo, la forma más fácil de realizar esta tarea es realizar las solicitudes una después de la otra. Esto tiene el inconveniente de que la segunda solicitud se iniciará solo cuando la primera haya finalizado. El tiempo total de ejecución será como minimo la suma de los dos tiempos de respuesta.

La solución a este problema, en un sistema síncrono, es comenzar hilos adicionales de control. Un *hilo* es otro programa activo cuya ejecución puede ser intercalada con otros programas por el sistema operativo—ya que la mayoría de las computadoras modernas contienen múltiples procesadores, múltiples hilos pueden incluso ejecutarse al mismo tiempo, en diferentes procesadores. Un segundo hilo podría iniciar la segunda solicitud, y luego ambos subprocesos esperan a que los resultados vuelvan, después de lo cual se vuelven a resincronizar para combinar sus resultados.

En el siguiente diagrama, las líneas gruesas representan el tiempo que el programa pasa corriendo normalmente, y las líneas finas representan el tiempo pasado esperando la red. En el modelo síncrono, el tiempo empleado por la red es *parte* de la línea de tiempo para un hilo de control dado. En el modelo asincrónico, comenzar una acción de red conceptualmente causa una *división* en la línea del tiempo. El programa que inició la acción continúa ejecutándose, y la acción ocurre junto a el, notificando al programa cuando está termina.



Otra forma de describir la diferencia es que esperar que las acciones terminen es *implicito* en el modelo síncrono, mientras que es *explicito*, bajo nuestro control, en el asincrónico.

La asincronicidad corta en ambos sentidos. Hace que expresar programas que hagan algo no se ajuste al modelo de control lineal más fácil, pero también puede hacer que expresar programas que siguen una línea recta sea más incómodo. Veremos algunas formas de abordar esta incomodidad más adelante en el capítulo.

Ambas de las plataformas de programación JavaScript importantes—navegadores y Node.js—realizan operaciones que pueden tomar un tiempo asincrónicamente, en lugar de confiar en hilos. Dado que la programación con hilos es notoria-

mente difícil (entender lo que hace un programa es mucho más difícil cuando está haciendo varias cosas a la vez), esto es generalmente considerado una buena cosa.

Tecnología cuervo

La mayoría de las personas son conscientes del hecho de que los cuervos son pájaros muy inteligentes. Pueden usar herramientas, planear con anticipación, recordar cosas e incluso comunicarse estas cosas entre ellos.

Lo que la mayoría de la gente no sabe, es que son capaces de hacer muchas cosas que mantienen bien escondidas de nosotros. Personas de buena reputación (un tanto excéntricas) expertas en córvidos, me han dicho que la tecnología cuervo no esta muy por detrás de la tecnología humana, y que nos estan alcanzando.

Por ejemplo, muchas culturas cuervo tienen la capacidad de construir dispositivos informáticos. Estos no son electrónicos, como lo son los dispositivos informáticos humanos, pero operan a través de las acciones de pequeños insectos, una especie estrechamente relacionada con las termitas, que ha desarrollado una relación simbiótica con los cuervos. Los pájaros les proporcionan comida, y a cambio los insectos construyen y operan sus complejas colonias que, con la ayuda de las criaturas vivientes dentro de ellos, realizan computaciones.

Tales colonias generalmente se encuentran en nidos grandes de larga vida. Las aves e insectos trabajan juntos para construir una red de estructuras bulbosas hechas de arcilla, escondidas entre las ramitas del nido, en el que los insectos viven y trabajan.

Para comunicarse con otros dispositivos, estas máquinas usan señales de luz. Los cuervos incrustan piezas de material reflectante en tallos de comunicación especial, y los insectos apuntan estos para reflejar la luz hacia otro nido, codificando los datos como una secuencia de flashes rápidos. Esto significa que solo los nidos que tienen una conexión visual ininterrumpida pueden comunicarse entre ellos.

Nuestro amigo, el experto en córvidos, ha mapeado la red de nidos de cuervo en el pueblo de Hières-sur-Amby, a orillas del río Ródano. Este mapa muestra los nidos y sus conexiones.

En un ejemplo asombroso de evolución convergente, las computadoras cuervo ejecutan JavaScript. En este capítulo vamos a escribir algunas funciones de redes básicas para ellos.

Devolución de llamadas

Un enfoque para la programación asincrónica es hacer que las funciones que realizan una acción lenta, tomen un argumento adicional, una *función de devolución de llamada*. La acción se inicia y, cuando esta finaliza, la función de devolución es llamada con el resultado.

Como ejemplo, la función `setTimeout`, disponible tanto en Node.js como en navegadores, espera una cantidad determinada de milisegundos (un segundo son mil milisegundos) y luego llama una función.

```
setTimeout(() => console.log("Tick"), 500);
```

Esperar no es generalmente un tipo de trabajo muy importante, pero puede ser útil cuando se hace algo como actualizar una animación o verificar si algo está tardando más que una cantidad dada de tiempo.

La realización de múltiples acciones asíncronas en una fila utilizando devoluciones de llamada significa que debes seguir pasando nuevas funciones para manejar la continuación de la computación después de las acciones.

La mayoría de las computadoras en los nidos de los cuervos tienen un bulbo de almacenamiento de datos a largo plazo, donde las piezas de información se graban en ramitas para que estas puedan ser recuperadas más tarde. Grabar o encontrar un fragmento de información requiere un momento, por lo que la interfaz para el almacenamiento a largo plazo es asíncrona y utiliza funciones de devolución de llamada.

Los bulbos de almacenamiento almacenan piezas de JSON-datos codificables

bajo nombres. Un cuervo podría almacenar información sobre los lugares donde hay comida escondida bajo el nombre `"caches de alimentos"`, que podría contener un array de nombres que apuntan a otros datos, que describen el caché real. Para buscar un caché de alimento en los bulbos de almacenamiento del nido *Gran Roble*, un cuervo podría ejecutar código como este:

```
import {granRoble} from "./tecnologia-cuervo";

granRoble.leerAlmacenamiento("caches de alimentos", caches => {
  let primerCache = caches[0];
  granRoble.leerAlmacenamiento(primerCache, informacion => {
    console.log(informacion);
  });
});
```

(Todos los nombres de las vinculaciones y los strings se han traducido del lenguaje cuervo a Español.)

Este estilo de programación es viable, pero el nivel de indentación aumenta con cada acción asincrónica, ya que terminas en otra función. Hacer cosas más complicadas, como ejecutar múltiples acciones al mismo tiempo, puede ser un poco incómodo.

Las computadoras cuervo están construidas para comunicarse usando pares de solicitud-respuesta. Eso significa que un nido envía un mensaje a otro nido, el cual inmediatamente envía un mensaje de vuelta, confirmando el recibo y, posiblemente, incluyendo una respuesta a una pregunta formulada en el mensaje.

Cada mensaje está etiquetado con un *tipo*, que determina cómo este es manejado. Nuestro código puede definir manejadores para tipos de solicitud específicos, y cuando se recibe una solicitud de este tipo, se llama al controlador para que este produzca una respuesta.

La interfaz exportada por el módulo `"./tecnologia-cuervo"` proporciona funciones de devolución de llamada para la comunicación. Los nidos tienen un método `enviar` que envía una solicitud. Este espera el nombre del nido objetivo, el tipo de solicitud y el contenido de la solicitud como sus primeros tres argumentos, y una función a llamar cuando llega una respuesta como su cuarto y último argumento.

```
granRoble.send("Pastura de Vacas", "nota", "Vamos a graznar fuerte a
    las 7PM",
           () => console.log("Nota entregada."));
```

Pero para hacer nidos capaces de recibir esa solicitud, primero tenemos que definir un tipo de solicitud llamado `"nota"`. El código que maneja las solicitudes debe ejecutarse no solo en este nido-computadora, sino en todos los nidos que puedan recibir mensajes de este tipo. Asumiremos que un cuervo sobrevuela e instala nuestro código controlador en todos los nidos.

```
import {definirTipoSolicitud} from "./tecnologia-cuervo";

definirTipoSolicitud("nota", (nido, contenido, fuente, listo) => {
  console.log(`${nido.nombre} recibio nota: ${contenido}`);
  listo();
});
```

La función `definirTipoSolicitud` define un nuevo tipo de solicitud. El ejemplo agrega soporte para solicitudes de tipo `"nota"`, que simplemente envían una nota a un nido dado. Nuestra implementación llama a `console.log` para que podamos verificar que la solicitud llegó. Los nidos tienen una propiedad `nombre` que contiene su nombre.

El cuarto argumento dado al controlador, `listo`, es una función de devolución de llamada que debe ser llamada cuando se finaliza con la solicitud. Si hubiesemos utilizado el valor de retorno del controlador como el valor de respuesta, eso significaria que un controlador de solicitud no puede realizar acciones asincrónicas por sí mismo. Una función que realiza trabajos asíncronos normalmente retorna antes de que el trabajo este hecho, habiendo arreglado que se llame una devolución de llamada cuando este completada. Entonces, necesitamos algún mecanismo asíncrono, en este caso, otra función de devolución de llamada—para indicar cuándo hay una respuesta disponible.

En cierto modo, la asincronía es *contagiosa*. Cualquier función que llame a una función que funcione asincrónicamente debe ser asíncrona en si misma, utilizando una devolución de llamada o algun mecanismo similar para entregar su resultado. Llamar devoluciones de llamada es algo más involucrado y propenso a errores que simplemente retornar un valor, por lo que necesitar estructurar grandes partes de tu programa de esa manera no es algo muy bueno.

Promesas

Trabajar con conceptos abstractos es a menudo más fácil cuando esos conceptos pueden ser representados por valores. En el caso de acciones asíncronas, podrías, en lugar de organizar a una función para que esta sea llamada en algún momento en el futuro, retornar un objeto que represente este evento en el

futuro.

Esto es para lo que es la clase estándar `Promise` ("Promesa"). Una *promesa* es una acción asíncrona que puede completarse en algún punto y producir un valor. Esta puede notificar a cualquier persona que esté interesada cuando su valor este disponible.

La forma más fácil de crear una promesa es llamando a `Promise.resolve` ("Promesa.resolver"). Esta función se asegura de que el valor que le des, sea envuelto en una promesa. Si ya es una promesa, simplemente es retornada—de lo contrario, obtienes una nueva promesa que termina de inmediato con tu valor como su resultado.

```
let quince = Promise.resolve(15);
quince.then(valor => console.log(`Obtuve ${valor}`));
// → Obtuve 15
```

Para obtener el resultado de una promesa, puede usar su método `then` ("entonces"). Este registra una (función de devolución de llamada) para que sea llamada cuando la promesa resuelva y produzca un valor. Puedes agregar múltiples devoluciones de llamada a una única promesa, y serán llamadas, incluso si las agregas después de que la promesa ya haya sido *resuelta* (terminada).

Pero eso no es todo lo que hace el método `then`. Este retorna otra promesa, que resuelve al valor que retorna la función del controlador o, si esa retorna una promesa, espera por esa promesa y luego resuelve su resultado.

Es útil pensar acerca de las promesas como dispositivos para mover valores a una realidad asincrónica. Un valor normal simplemente esta allí. Un valor prometido es un valor que *podría* ya estar allí o podría aparecer en algún momento en el futuro. Las computaciones definidas en términos de promesas actúan en tales valores envueltos y se ejecutan de forma asíncrona a medida los valores se vuelven disponibles.

Para crear una promesa, puedes usar `Promise` como un constructor. Tiene una interfaz algo extraña—el constructor espera una función como argumento, a la cual llama inmediatamente, pasando una función que puede usar para resolver la promesa. Funciona de esta manera, en lugar de, por ejemplo, con un método `resolve`, de modo que solo el código que creó la promesa pueda resolverla.

Así es como crearía una interfaz basada en promesas para la función `leerAlmacenamiento`
.

```
function almacenamiento(nido, nombre) {
  return new Promise(resolve => {
```

```
    nido.leerAlmacenamiento(nombre, resultado => resolve(resultado))
        ;
  });
}

almacenamiento(granRoble, "enemigos")
  .then(valor => console.log("Obtuve", valor));
```

Esta función asíncrona retorna un valor significativo. Esta es la principal ventaja de las promesas—simplifican el uso de funciones asincrónicas. En lugar de tener que pasar devoluciones de llamadas, las funciones basadas en promesas son similares a las normales: toman entradas como argumentos y retornan su resultado. La única diferencia es que la salida puede que no este disponible inmediatamente.

Fracaso

Las computaciones regulares en JavaScript pueden fallar lanzando una excepción. Las computaciones asincrónicas a menudo necesitan algo así. Una solicitud de red puede fallar, o algún código que sea parte de la computación asincrónica puede arrojar una excepción.

Uno de los problemas más urgentes con el estilo de devolución de llamadas en la programación asíncrona es que hace que sea extremadamente difícil asegurarte de que las fallas sean reportadas correctamente a las devoluciones de llamada.

Una convención ampliamente utilizada es que el primer argumento para la devolución de llamada es usado para indicar que la acción falló, y el segundo contiene el valor producido por la acción cuando tuvo éxito. Tales funciones de devolución de llamadas siempre deben verificar si recibieron una excepción, y asegurarse de que cualquier problema que causen, incluidas las excepciones lanzadas por las funciones que estas llaman, sean atrapadas y entregadas a la función correcta.

Las promesas hacen esto más fácil. Estas pueden ser resueltas (la acción termino con éxito) o rechazadas (esta falló). Los controladores de resolución (registrados con `then`) solo se llaman cuando la acción es exitosa, y los rechazos se propagan automáticamente a la nueva promesa que es retornada por `then`. Y cuando un controlador arroje una excepción, esto automáticamente hace que la promesa producida por su llamada `then` sea rechazada. Entonces, si cualquier elemento en una cadena de acciones asíncronas falla, el resultado de toda la

cadena se marca como rechazado, y no se llaman más manejadores despues del punto en donde falló.

Al igual que resolver una promesa proporciona un valor, rechazar una también proporciona uno, generalmente llamado la *razón* el rechazo. Cuando una excepción en una función de controlador provoca el rechazo, el valor de la excepción se usa como la razón. Del mismo modo, cuando un controlador retorna una promesa que es rechazada, ese rechazo fluye hacia la próxima promesa. Hay una función `Promise.reject` que crea una nueva promesa inmediatamente rechazada.

Para manejar explícitamente tales rechazos, las promesas tienen un método `catch` ("atraoar") que registra un controlador para que sea llamado cuando se rechaze la promesa, similar a cómo los manejadores `then` manejan la resolución normal. También es muy parecido a `then` en que retorna una nueva promesa, que se resuelve en el valor de la promesa original si esta se resuelve normalmente, y al resultado del controlador `catch` de lo contrario. Si un controlador `catch` lanza un error, la nueva promesa también es rechazada.

Como una abreviatura, `then` también acepta un manejador de rechazo como segundo argumento, por lo que puedes instalar ambos tipos de controladores en un solo método de llamada.

Una función que se pasa al constructor `Promise` recibe un segundo argumento, junto con la función de resolución, que puede usar para rechazar la nueva promesa.

Las cadenas de promesas creadas por llamadas a `then` y `catch` puede verse como una tubería a través de la cual los valores asíncronicos o las fallas se mueven. Dado que tales cadenas se crean mediante el registro de controladores, cada enlace tiene un controlador de éxito o un controlador de rechazo (o ambos) asociados a ello. Controladores que no coinciden con ese tipo de resultados (éxito o fracaso) son ignorados. Pero los que sí coinciden son llamados, y su resultado determina qué tipo de valor viene después—éxito cuando retorna un valor que no es una promesa, rechazo cuando arroja una excepción, y el resultado de una promesa cuando retorna una de esas.

Al igual que una excepción no detectada es manejada por el entorno, Los entornos de JavaScript pueden detectar cuándo una promesa rechazada no es manejada, y reportará esto como un error.

Las redes son difíciles

Ocasionalmente, no hay suficiente luz para los sistemas de espejos de los cuervos para transmitir una señal, o algo bloquea el camino de la señal. Es posible que

se envíe una señal, pero que nunca se reciba.

Tal y como es, eso solo causará que la devolución de llamada dada a `send` nunca sea llamada, lo que probablemente hará que el programa se detenga sin siquiera notar que hay un problema. Sería bueno si, después de un determinado período de no obtener una respuesta, una solicitud *expirará* e informara de un fracaso.

A menudo, las fallas de transmisión son accidentes aleatorios, como la luz del faro de un auto interfieriendo con las señales de luz, y simplemente volver a intentar la solicitud puede hacer que esta tenga éxito. Entonces, mientras estamos en eso, hagamos que nuestra función de solicitud automáticamente reintente el envío de la solicitud momentos antes de que se de por vencida.

Y, como hemos establecido que las promesas son algo bueno, tambien haremos que nuestra función de solicitud retorne una promesa. En términos de lo que pueden expresar, las devoluciones de llamada y las promesas son equivalentes. Las funciones basadas en devoluciones de llamadas se pueden envolver para exponer una interfaz basada en promesas, y viceversa.

Incluso cuando una solicitud y su respuesta sean entregadas exitosamente, la respuesta puede indicar un error—por ejemplo, si la solicitud intenta utilizar un tipo de solicitud que no haya sido definida o si el controlador genera un error. Para soportar esto, `send` y `definirTipoSolicitud` siguen la convención mencionada anteriormente, donde el primer argumento pasado a las devoluciones de llamada es el motivo del fallo, si lo hay, y el segundo es el resultado real.

Estos pueden ser traducidos para prometer resolución y rechazo por parte de nuestra envoltura.

```
class TiempoDeEspera extends Error {}

function request(nido, objetivo, tipo, contenido) {
  return new Promise((resolve, reject) => {
    let listo = false;
    function intentar(n) {
      nido.send(objetivo, tipo, contenido, (fallo, value) => {
        listo = true;
        if (fallo) reject(fallo);
        else resolve(value);
      });
      setTimeout(() => {
        if (listo) return;
        else if (n < 3) intentar(n + 1);
        else reject(new TiempoDeEspera("Tiempo de espera agotado"));
      }, 250);
```

```
    }
    intentar(1);
  });
}
```

Debido a que las promesas solo se pueden resolver (o rechazar) una vez, esto funcionara. La primera vez que se llame a `resolve` o `reject` se determinara el resultado de la promesa y cualquier llamada subsecuente, como el tiempo de espera que llega después de que finaliza la solicitud, o una solicitud que regresa después de que otra solicitud es finalizada, es ignorada.

Para construir un ciclo asincrónico, para los reintentos, necesitamos usar un función recursiva—un ciclo regular no nos permite detenernos y esperar por una acción asincrónica. La función `intentar` hace un solo intento de enviar una solicitud. También establece un tiempo de espera que, si no ha regresado una respuesta después de 250 milisegundos, comienza el próximo intento o, si este es el cuarto intento, rechaza la promesa con una instancia de `TiempoDeEspera` como la razón.

Volver a intentar cada cuarto de segundo y rendirse cuando no ha llegado ninguna respuesta después de un segundo es algo definitivamente arbitrario. Es incluso posible, si la solicitud llegó pero el controlador se esta tardando un poco más, que las solicitudes se entreguen varias veces. Escribiremos nuestros manejadores con ese problema en mente—los mensajes duplicados deberían de ser inofensivos.

En general, no construiremos una red robusta de clase mundial hoy. Pero eso esta bien—los cuervos no tienen expectativas muy altas todavía cuando se trata de la computación.

Para aislarnos por completo de las devoluciones de llamadas, seguiremos adelante y también definiremos un contenedor para `definirTipoSolicitud` que permite que la función controlador pueda retornar una promesa o valor normal, y envia eso hasta la devolución de llamada para nosotros.

```
function tipoSolicitud(nombre, manejador) {
  definirTipoSolicitud(nombre, (nido, contenido, fuente,
                           devolucionDeLlamada) => {
    try {
      Promise.resolve(manejador(nido, contenido, fuente))
        .then(response => devolucionDeLlamada(null, response),
              failure => devolucionDeLlamada(failure));
    } catch (exception) {
      devolucionDeLlamada(exception);
    }
```

```
  });
}
```

`Promise.resolve` se usa para convertir el valor retornado por `manejador` a una promesa si no es una ya.

Ten en cuenta que la llamada a `manejador` tenía que estar envuelta en un bloque `try`, para asegurarse de que cualquier excepción que aparezca directamente se le dé a la devolución de llamada. Esto ilustra muy bien la dificultad de manejar adecuadamente los errores con devoluciones de llamada crudas—es muy fácil olvidarse de encaminar correctamente excepciones como esa, y si no lo haces, las fallas no se seran informadas a la devolución de llamada correcta. Las promesas hacen esto casi automático, y por lo tanto, son menos propensas a errores.

Colecciones de promesas

Cada computadora nido mantiene un array de otros nidos dentro de la distancia de transmisión en su propiedad `vecinos`. Para verificar cuáles de esos son actualmente accesibles, puede escribir una función que intente enviar un solicitud `"ping"` (una solicitud que simplemente pregunta por una respuesta) para cada de ellos, y ver cuáles regresan.

Al trabajar con colecciones de promesas que se ejecutan al mismo tiempo, la función `Promise.all` puede ser útil. Esta retorna una promesa que espera a que se resuelvan todas las promesas del array, y luego resuelve un array de los valores que estas promesas produjeron (en el mismo orden que en el array original). Si alguna promesa es rechazada, el el resultado de `Promise.all` es en sí mismo rechazado.

```
tipoSolicitud("ping", () => "pong");

function vecinosDisponibles(nido) {
  let solicitudes = nido.vecinos.map(vecino => {
    return request(nido, vecino, "ping")
      .then(() => true, () => false);
  });
  return Promise.all(solicitudes).then(resultado => {
    return nido.vecinos.filter((_, i) => resultado[i]);
  });
}
```

Cuando un vecino no este disponible, no queremos que todo la promesa combinada falle, dado que entonces no sabríamos nada. Entonces la función que es mappeada en el conjunto de vecinos para convertirlos en promesas de solicitud vincula a los controladores que hacen las solicitudes exitosas produzcan `true` y las rechazadas produzcan `false`.

En el controlador de la promesa combinada, `filter` se usa para eliminar esos elementos de la matriz `vecinos` cuyo valor correspondiente es falso. Esto hace uso del hecho de que `filter` pasa el índice de matriz del elemento actual como segundo argumento para su función de filtrado (`map`,`some`, y métodos similares de orden superior de arrays hacen lo mismo).

Inundación de red

El hecho de que los nidos solo pueden hablar con sus vecinos inhibe en gran cantidad la utilidad de esta red.

Para transmitir información a toda la red, una solución es configurar un tipo de solicitud que sea reenviada automáticamente a los vecinos. Estos vecinos luego la envían a sus vecinos, hasta que toda la red ha recibido el mensaje.

```
import {todosLados} from "./tecnologia-cuervo";

todosLados(nido => {
  nido.estado.chismorreo = [];
});

function enviarChismorreo(nido, mensaje, exceptoPor = null) {
  nido.estado.chismorreo.push(mensaje);
  for (let vecino of nido.vecinos) {
    if (vecino == exceptoPor) continue;
    request(nido, vecino, "chismorreo", mensaje);
  }
}

requestType("chismorreo", (nido, mensaje, fuente) => {
  if (nido.estado.chismorreo.includes(mensaje)) return;
  console.log(`${nido.nombre} recibio chismorreo '${
               mensaje}' de ${fuente}`);
  enviarChismorreo(nido, mensaje, fuente);
});
```

Para evitar enviar el mismo mensaje a traves de la red por siempre, cada nido mantiene un array de strings de chismorreos que ya ha visto. Para definir

este array, usaremos la función `todosLados`—que ejecuta código en todos los nidos—para añadir una propiedad al objeto `estado` del nido, que es donde mantendremos estado local del nido.

Cuando un nido recibe un mensaje de chisme duplicado, lo cual es muy probable que suceda con todo el mundo reenviando estos a ciegas, lo ignora. Pero cuando recibe un mensaje nuevo, emocionadamente le dice a todos sus vecinos a excepción de quien le envió el mensaje.

Esto provocará que una nueva pieza de chismes se propague a través de la red como una mancha de tinta en agua. Incluso cuando algunas conexiones no estan trabajando actualmente, si hay una ruta alternativa a un nido dado, el chisme llegará hasta allí.

Este estilo de comunicación de red se llama *inundamiento*-inunda la red con una pieza de información hasta que todos los nodos la tengan.

Enrutamiento de mensajes

Si un nodo determinado quiere hablar unicamente con otro nodo, la inundación no es un enfoque muy eficiente. Especialmente cuando la red es grande, daría lugar a una gran cantidad de transferencias de datos inútiles.

Un enfoque alternativo es configurar una manera en que los mensajes salten de nodo a nodo, hasta que lleguen a su destino. La dificultad con eso es que requiere de conocimiento sobre el diseño de la red. Para enviar una solicitud hacia la dirección de un nido lejano, es necesario saber qué nido vecino lo acerca más a su destino. Enviar la solicitud en la dirección equivocada no servirá de mucho.

Dado que cada nido solo conoce a sus vecinos directos, no tiene la información que necesita para calcular una ruta. De alguna manera debemos extender la información acerca de estas conexiones a todos los nidos. Preferiblemente en una manera que permita ser cambiada con el tiempo, cuando los nidos son abandonados o nuevos nidos son construidos.

Podemos usar la inundación de nuevo, pero en lugar de verificar si un determinado mensaje ya ha sido recibido, ahora verificamos si el nuevo conjunto de vecinos de un nido determinado coinciden con el conjunto actual que tenemos para él.

```
tipoSolicitud("conexiones", (nido, {nombre, vecinos},
                             fuente) => {
  let conexiones = nido.estado.conexiones;
  if (JSON.stringify(conexiones.get(nombre)) ==
      JSON.stringify(vecinos)) return;
```

```
  conexiones.set(nombre, vecinos);
  difundirConexiones(nido, nombre, fuente);
});

function difundirConexiones(nido, nombre, exceptoPor = null) {
  for (let vecino of nido.vecinos) {
    if (vecino == exceptoPor) continue;
    solicitud(nido, vecino, "conexiones", {
      nombre,
      vecinos: nido.estado.conexiones.get(nombre)
    });
  }
}

todosLados(nido => {
  nido.estado.conexiones = new Map;
  nido.estado.conexiones.set(nido.nombre, nido.vecinos);
  difundirConexiones(nido, nido.nombre);
});
```

La comparación usa `JSON.stringify` porque `==`, en objetos o arrays, solo retornara true cuando los dos tengan exactamente el mismo valor, lo cual no es lo que necesitamos aquí. Comparar los strings JSON es una cruda pero efectiva manera de comparar su contenido.

Los nodos comienzan inmediatamente a transmitir sus conexiones, lo que debería, a menos que algunos nidos sean completamente inalcanzables, dar rápidamente cada nido un mapa del grafo de la red actual.

Una cosa que puedes hacer con grafos es encontrar rutas en ellos, como vimos en el Capítulo 7. Si tenemos una ruta hacia el destino de un mensaje, sabemos en qué dirección enviarlo.

Esta función `encontrarRuta`, que se parece mucho a `encontrarRuta` del Capítulo 7, busca por una forma de llegar a un determinado nodo en la red. Pero en lugar de devolver toda la ruta, simplemente retorna el siguiente paso. Ese próximo nido en si mismo, usando su información actual sobre la red, decididira *hacia* dónde enviar el mensaje.

```
function encontrarRuta(desde, hasta, conexiones) {
  let trabajo = [{donde: desde, via: null}];
  for (let i = 0; i < trabajo.length; i++) {
    let {donde, via} = trabajo[i];
    for (let siguiente of conexiones.get(donde) || []) {
      if (siguiente == hasta) return via;
      if (!trabajo.some(w => w.donde == siguiente)) {
```

```
        trabajo.push({donde: siguiente, via: via || siguiente});
      }
    }
  }
  return null;
}
```

Ahora podemos construir una función que pueda enviar mensajes de larga distancia. Si el mensaje está dirigido a un vecino directo, se entrega normalmente. Si no, se empaqueta en un objeto y se envía a un vecino que este más cerca del objetivo, usando el tipo de solicitud `"ruta"`, que hace que ese vecino repita el mismo comportamiento.

```
function solicitudRuta(nido, objetivo, tipo, contenido) {
  if (nido.vecinos.includes(objetivo)) {
    return solicitud(nido, objetivo, tipo, contenido);
  } else {
    let via = encontrarRuta(nido.nombre, objetivo,
                            nido.estado.conexiones);
    if (!via) throw new Error(`No hay rutas disponibles hacia ${
       objetivo}`);
    return solicitud(nido, via, "ruta",
                     {objetivo, tipo, contenido});
  }
}

tipoSolicitud("ruta", (nido, {objetivo, tipo, contenido}) => {
  return solicitudRuta(nido, objetivo, tipo, contenido);
});
```

Hemos construido varias capas de funcionalidad sobre un sistema de comunicación primitivo para que sea conveniente de usarlo. Este es un buen (aunque simplificado) modelo de cómo las redes de computadoras reales trabajan.

Una propiedad distintiva de las redes de computadoras es que no son confiables—las abstracciones construidas encima de ellas pueden ayudar, pero no se puede abstraer la falla de una falla de red. Entonces la programación de redes es típicamente mucho acerca de anticipar y lidiar con fallas.

Funciones asíncronas

Para almacenar información importante, se sabe que los cuervos la duplican a través de los nidos. De esta forma, cuando un halcón destruye un nido, la información no se pierde.

Para obtener una pieza de información dada que no este en su propia bulbo de almacenamiento, una computadora nido puede consultar otros nidos al azar en la red hasta que encuentre uno que la tenga.

```
tipoSolicitud("almacenamiento", (nido, nombre) => almacenamiento(
   nido, nombre));

function encontrarEnAlmacenamiento(nido, nombre) {
  return almacenamiento(nido, nombre).then(encontrado => {
    if (encontrado != null) return encontrado;
    else return encontrarEnAlmacenamientoRemoto(nido, nombre);
  });
}

function red(nido) {
  return Array.from(nido.estado.conexiones.keys());
}

function encontrarEnAlmacenamientoRemoto(nido, nombre) {
  let fuentes = red(nido).filter(n => n != nido.nombre);
  function siguiente() {
    if (fuentes.length == 0) {
      return Promise.reject(new Error("No encontrado"));
    } else {
      let fuente = fuentes[Math.floor(Math.random() *
                                      fuentes.length)];
      fuentes = fuentes.filter(n => n != fuente);
      return solicitudRuta(nido, fuente, "almacenamiento", nombre)
        .then(valor => valor != null ? valor : siguiente(),
              siguiente);
    }
  }
  return siguiente();
}
```

Como `conexiones` es un `Map`, `Object.keys` no funciona en él. Este tiene un *metódo* `keys`, pero que retorna un iterador en lugar de un array. Un iterador (o valor iterable) se puede convertir a un array con la función `Array.from`.

Incluso con promesas, este es un código bastante incómodo. Múltiples ac-

ciones asincrónicas están encadenadas juntas de maneras no-obvias. Nosotros de nuevo necesitamos una función recursiva (`siguiente`) para modelar ciclos a través de nidos.

Y lo que el código realmente hace es completamente lineal—siempre espera a que se complete la acción anterior antes de comenzar la siguiente. En un modelo de programación sincrónica, sería más simple de expresar.

La buena noticia es que JavaScript te permite escribir código pseudo-sincrónico. Una función `async` es una función que retorna implícitamente una promesa y que puede, en su cuerpo, `await` ("esperar") otras promesas de una manera que *se ve* sincrónica.

Podemos reescribir `encontrarEnAlmacenamiento` de esta manera:

```
async function encontrarEnAlmacenamiento(nido, nombre) {
  let local = await almacenamiento(nido, nombre);
  if (local != null) return local;

  let fuentes = red(nido).filter(n => n != nido.nombre);
  while (fuentes.length > 0) {
    let fuente = fuentes[Math.floor(Math.random() *
                                    fuentes.length)];
    fuentes = fuentes.filter(n => n != fuente);
    try {
      let encontrado = await solicitudRuta(nido, fuente, "
          almacenamiento",
                                           nombre);
      if (encontrado != null) return encontrado;
    } catch (_) {}
  }
  throw new Error("No encontrado");
}
```

Una función `async` está marcada por la palabra `async` antes de la palabra clave `function`. Los métodos también pueden hacerse `async` al escribir `async` antes de su nombre. Cuando se llame a dicha función o método, este retorna una promesa. Tan pronto como el cuerpo retorne algo, esa promesa es resuelta Si arroja una excepción, la promesa es rechazada.

Dentro de una función `async`, la palabra `await` se puede poner delante de una expresión para esperar a que se resuelva una promesa, y solo entonces continua la ejecución de la función.

Tal función ya no se ejecuta, como una función regular de JavaScript de principio a fin de una sola vez. En su lugar, puede ser *congelada* en cualquier punto que tenga un `await`, y se reanuda en un momento posterior.

Para código asincrónico no-trivial, esta notación suele ser más conveniente que usar promesas directamente. Incluso si necesitas hacer algo que no se ajuste al modelo síncrono, como realizar múltiples acciones al mismo tiempo, es fácil combinar `await` con el uso directo de promesas.

Generadores

Esta capacidad de las funciones para pausar y luego reanudarse nuevamente no es exclusiva para las funciones `async`. JavaScript también tiene una característica llamada funciones *generador*. Estss son similares, pero sin las promesas.

Cuando defines una función con `function*` (colocando un asterisco después de la palabra `function`), se convierte en un generador. Cuando llamas un generador, este retorna un iterador, que ya vimos en el Capítulo 6.

```
function* potenciacion(n) {
  for (let actual = n;; actual *= n) {
    yield actual;
  }
}

for (let potencia of potenciacion(3)) {
  if (potencia > 50) break;
  console.log(potencia);
}
// → 3
// → 9
// → 27
```

Inicialmente, cuando llamas a `potenciacion`, la función se congela en su comienzo. Cada vez que llames `next` en el iterador, la función se ejecuta hasta que encuentre una expresión `yield` ("arrojar"), que la pausa y causa que el valor arrojado se convierta en el siguiente valor producido por el iterador. Cuando la función retorne (la del ejemplo nunca lo hace), el iterador está completo.

Escribir iteradores es a menudo mucho más fácil cuando usas funciones generadoras. El iterador para la clase grupal (del ejercicio en el Capítulo 6) se puede escribir con este generador:

```
Conjunto.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.miembros.length; i++) {
    yield this.miembros[i];
  }
};
```

Ya no es necesario crear un objeto para mantener el estado de la iteración—los generadores guardan automáticamente su estado local cada vez ellos arrojen.

Dichas expresiones `yield` solo pueden ocurrir directamente en la función generadora en sí y no en una función interna que definas dentro de ella. El estado que ahorra un generador, cuando arroja, es solo su entorno *local* y la posición en la que fue arrojada.

Una función `async` es un tipo especial de generador. Produce una promesa cuando se llama, que se resuelve cuando vuelve (termina) y rechaza cuando arroja una excepción. Cuando cede (espera) por una promesa, el resultado de esa promesa (valor o excepción lanzada) es el resultado de la expresión `await`.

El ciclo de evento

Los programas asincrónicos son ejecutados pieza por pieza. Cada pieza puede iniciar algunas acciones y programar código para que se ejecute cuando la acción termine o falle. Entre estas piezas, el programa permanece inactivo, esperando por la siguiente acción.

Por lo tanto, las devoluciones de llamada no son llamadas directamente por el código que las programó. Si llamo a `setTimeout` desde adentro de una función, esa función habra retornado para el momento en que se llame a la función de devolución de llamada. Y cuando la devolución de llamada retorne, el control no volvera a la función que la programo.

El comportamiento asincrónico ocurre en su propia función de llamada de pila vacía. Esta es una de las razones por las cuales, sin promesas, la gestión de excepciones en el código asincrónico es dificil. Como cada devolución de llamada comienza con una pila en su mayoría vacía, tus manejadores `catch` no estarán en la pila cuando lanzen una excepción.

```
try {
  setTimeout(() => {
    throw new Error("Woosh");
  }, 20);
} catch (_) {
  // Esto no se va a ejecutar
  console.log("Atrapado!");
}
```

No importa que tan cerca los eventos—como tiempos de espera o solicitudes entrantes—sucedan, un entorno de JavaScript solo ejecutará un programa a

la vez. Puedes pensar en esto como un gran ciclo *alrededor* de tu programa, llamado *ciclo de evento*. Cuando no hay nada que hacer, ese bucle está detenido. Pero a medida que los eventos entran, se agregan a una cola, y su código se ejecuta uno después del otro. Porque no hay dos cosas que se ejecuten al mismo tiempo, código de ejecución lenta puede retrasar el manejo de otros eventos.

Este ejemplo establece un tiempo de espera, pero luego se retrasa hasta después del tiempo de espera previsto, lo que hace que el tiempo de espera este tarde.

```
let comienzo = Date.now();
setTimeout(() => {
  console.log("Tiempo de espera corrio al ", Date.now() - comienzo);
}, 20);
while (Date.now() < comienzo + 50) {}
console.log("Se desperdicio tiempo hasta el ", Date.now() - comienzo
   );
// → Se desperdicio tiempo hasta el 50
// → Tiempo de espera corrio al 55
```

Las promesas siempre se resuelven o rechazan como un nuevo evento. Incluso si una promesa ya ha sido resuelta, esperar por ella hará que la devolución de llamada se ejecute después de que el script actual termine, en lugar de hacerlo inmediatamente.

```
Promise.resolve("Listo").then(console.log);
console.log("Yo primero!");
// → Yo primero!
// → Listo
```

En capítulos posteriores, veremos otros tipos de eventos que se ejecutan en el ciclo de eventos.

Errores asincrónicos

Cuando tu programa se ejecuta de forma síncrona, de una sola vez, no hay cambios de estado sucediendo aparte de aquellos que el mismo programa realiza. Para los programas asíncronos, esto es diferente—estos pueden tener *brechas* en su ejecución durante las cuales se podria ejecutar otro código.

Veamos un ejemplo. Uno de los pasatiempos de nuestros cuervos es contar la cantidad de polluelos que nacen en el pueblo cada año. Los nidos guardan este recuento en sus bulbos de almacenamiento. El siguiente código intenta

enumerar los recuentos de todos los nidos para un año determinado.

```
function cualquierAlmacenamiento(nido, fuente, nombre) {
  if (fuente == nido.nombre) return almacenamiento(nido, nombre);
  else return solicitudRuta(nido, fuente, "almacenamiento", nombre);
}

async function polluelos(nido, años) {
  let lista = "";
  await Promise.all(red(nido).map(async nombre => {
    lista += `${nombre}: ${
      await cualquierAlmacenamiento(nido, nombre, `polluelos en ${
         años}`)
    }\n`;
  }));
  return lista;
}
```

La parte `async nombre =>` muestra que las funciones de flecha también pueden ser `async` al poner la palabra `async` delante de ellas.

El código no parece sospechoso de inmediato... mapea la función de flecha `async` sobre el conjunto de nidos, creando una serie de promesas, y luego usa `Promise.all` para esperar a todos estas antes de retornar la lista que estas construyen.

Pero está seriamente roto. Siempre devolverá solo una línea de salida, enumerando al nido que fue más lento en responder.

Puedes averiguar por qué?

El problema radica en el operador `+=`, que toma el valor *actual* de `lista` en el momento en que la instrucción comienza a ejecutarse, y luego, cuando el `await` termina, establece que la vinculaciòn `lista` sea ese valor más el string agregado.

Pero entre el momento en el que la declaración comienza a ejecutarse y el momento donde termina hay una brecha asincrónica. La expresión `map` se ejecuta antes de que se haya agregado algo a la lista, por lo que cada uno de los operadores `+=` comienza desde un string vacío y termina cuando su recuperación de almacenamiento finaliza, estableciendo `lista` como una lista de una sola línea—el resultado de agregar su línea al string vacío.

Esto podría haberse evitado fácilmente retornando las líneas de las promesas mapeadas y llamando a `join` en el resultado de `Promise.all`, en lugar de construir la lista cambiando una vinculación. Como siempre, calcular nuevos valores es menos propenso a errores que cambiar valores existentes.

```
async function polluelos(nido, año) {
  let lineas = red(nido).map(async nombre => {
    return nombre + ": " +
      await cualquierAlmacenamiento(nido, nombre, `polluelos en ${
          año}`);
  });
  return (await Promise.all(lineas)).join("\n");
}
```

Errores como este son fáciles de hacer, especialmente cuando se usa `await`, y debes tener en cuenta dónde se producen las brechas en tu código. Una ventaja de la asincronicidad *explicita* de JavaScript (ya sea a través de devoluciones de llamada, promesas, o `await`) es que detectar estas brechas es relativamente fácil.

Resumen

La programación asincrónica permite expresar la espera de acciones de larga duración sin congelar el programa durante estas acciones. Los entornos de JavaScript suelen implementar este estilo de programación usando devoluciones de llamada, funciones que son llaman cuando las acciones son completadas. Un ciclo de eventos planifica que dichas devoluciones de llamadas sean llamadas cuando sea apropiado, una después de la otra, para que sus ejecuciones no se superpongan.

La programación asíncrona se hace más fácil mediante promesas, objetos que representar acciones que podrían completarse en el futuro, y funciones `async`, que te permiten escribir un programa asíncrono como si fuera sincrónico.

Ejercicios

Siguiendo el bisturí

Los cuervos del pueblo poseen un viejo bisturí que ocasionalmente usan en misiones especiales—por ejemplo, para cortar puertas de malla o embalar cosas. Para ser capaces de rastrearlo rápidamente, cada vez que se mueve el bisturí a otro nido, una entrada se agrega al almacenamiento tanto del nido que lo tenía como al nido que lo tomó, bajo el nombre `"bisturí"`, con su nueva ubicación como su valor.

Esto significa que encontrar el bisturí es una cuestión de seguir la ruta de navegación de las entradas de almacenamiento, hasta que encuentres un nido

que apunte a el nido en si mismo.

Escribe una función `async`, `localizarBisturi` que haga esto, comenzando en el nido en el que se ejecute. Puede usar la función `cualquierAlmacenamiento` definida anteriormente para acceder al almacenamiento en nidos arbitrarios. El bisturí ha estado dando vueltas el tiempo suficiente como para que puedas suponer que cada nido tiene una entrada `bisturí` en su almacenamiento de datos.

Luego, vuelve a escribir la misma función sin usar `async` y `await`.

Las fallas de solicitud se muestran correctamente como rechazos de la promesa devuelta en ambas versiones? Cómo?

Construyendo Promise.all

Dado un array de promesas, `Promise.all` retorna una promesa que espera a que finalicen todas las promesas del array. Entonces tiene éxito, produciendo un array de valores de resultados. Si una promesa en el array falla, la promesa retornada por `all` también falla, con la razón de la falla proveniente de la promesa fallida.

Implemente algo como esto tu mismo como una función regular llamada `Promise_all`.

Recuerda que una vez que una promesa ha tenido éxito o ha fallado, no puede tener éxito o fallar de nuevo, y llamadas subsecuentes a las funciones que resuelven son ignoradas. Esto puede simplificar la forma en que manejas la falla de tu promesa.

"El evaluador, que determina el significado de las expresiones en un lenguaje de programación, es solo otro programa."

—Hal Abelson and Gerald Sussman, Structure and Interpretation of Computer Programs

Chapter 12

Proyecto: Un Lenguaje de Programación

Construir tu propio lenguaje de programación es sorprendentemente fácil (siempre y cuando no apuntes demasiado alto) y muy esclarecedor.

Lo principal que quiero mostrar en este capítulo es que no hay magia involucrada en la construcción de tu propio lenguaje. A menudo he sentido que algunos inventos humanos eran tan inmensamente inteligentes y complicados que nunca podría llegar a entenderlos. Pero con un poco de lectura y experimentación, a menudo resultan ser bastante mundanos.

Construiremos un lenguaje de programación llamado Egg. Será un lenguaje pequeño y simple—pero lo suficientemente poderoso como para expresar cualquier computación que puedes pensar. Permitirá una abstracción simple basada en funciones.

Análisis

La parte más visible de un lenguaje de programación es su *sintaxis*, o notación. Un *analizador* es un programa que lee una pieza de texto y produce una estructura de datos que refleja la estructura del programa contenido en ese texto. Si el texto no forma un programa válido, el analizador debe señalar el error.

Nuestro lenguaje tendrá una sintaxis simple y uniforme. Todo en Egg es una expresión. Una expresión puede ser el nombre de una vinculación (binding), un número, una cadena de texto o una *aplicación*. Las aplicaciones son usadas para llamadas de función pero también para constructos como `if` o `while`.

Para mantener el analizador simple, las cadenas en Egg no soportarán nada parecido a escapes de barra invertida. Una cadena es simplemente una secuencia de caracteres que no sean comillas dobles, envueltas en comillas dobles. Un número es un secuencia de dígitos. Los nombres de vinculaciones pueden consistir en cualquier carácter no que sea un espacio en blanco y eso no tiene un significado especial en la sintaxis.

Las aplicaciones se escriben tal y como están en JavaScript, poniendo paréntesis después de una expresión y teniendo cualquier cantidad de argumentos

entre esos paréntesis, separados por comas.

```
hacer(definir(x, 10),
   si(>(x, 5),
      imprimir("grande"),
      imprimir("pequeño")))
```

La uniformidad del lenguaje Egg significa que las cosas que son operadores en JavaScript (como `>`) son vinculaciones normales en este lenguaje, aplicadas como cualquier función. Y dado que la sintaxis no tiene un concepto de bloque, necesitamos una construcción `hacer` para representar el hecho de realizar múltiples cosas en secuencia.

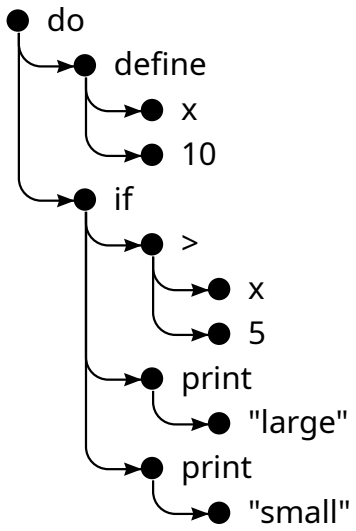
La estructura de datos que el analizador usará para describir un programa consta de objetos de expresión, cada uno de los cuales tiene una propiedad `tipo` que indica el tipo de expresión que este es y otras propiedades que describen su contenido.

Las expresiones de tipo `"valor"` representan strings o números literales. Su propiedad `valor` contiene el string o valor numérico que estos representan. Las expresiones de tipo `"palabra"` se usan para identificadores (nombres). Dichos objetos tienen una propiedad `nombre` que contienen el nombre del identificador como un string. Finalmente, las expresiones `"aplicar"` representan aplicaciones. Tienen una propiedad `operador` que se refiere a la expresión que está siendo aplicada, y una propiedad `argumentos` que contiene un array de expresiones de argumentos.

La parte `>(x, 5)` del programa anterior se representaría de esta manera:

```
{
  tipo: "aplicar",
  operador: {tipo: "palabra", nombre: ">"},
  argumentos: [
    {tipo: "palabra", nombre: "x"},
    {tipo: "valor", valor: 5}
  ]
}
```

Tal estructura de datos se llama *árbol de sintaxis*. Si tu imaginas los objetos como puntos y los enlaces entre ellos como líneas entre esos puntos, tiene una forma similar a un árbol. El hecho de que las expresiones contienen otras expresiones, que a su vez pueden contener más expresiones, es similar a la forma en que las ramas de los árboles se dividen y dividen una y otra vez.



Compara esto con el analizador que escribimos para el formato de archivo de configuración en el Capítulo 9, que tenía una estructura simple: dividía la entrada en líneas y manejaba esas líneas una a la vez. Habían solo unas pocas formas simples que se le permitía tener a una línea.

Aquí debemos encontrar un enfoque diferente. Las expresiones no están separadas en líneas, y tienen una estructura recursiva. Las expresiones de aplicaciones *contienen* otras expresiones.

Afortunadamente, este problema se puede resolver muy bien escribiendo una función analizadora que es recursiva en una manera que refleje la naturaleza recursiva del lenguaje.

Definimos una función `analizarExpresion`, que toma un string como entrada y retorna un objeto que contiene la estructura de datos para la expresión al comienzo del string, junto con la parte del string que queda después de analizar esta expresión Al analizar subexpresiones (el argumento para una aplicación, por ejemplo), esta función puede ser llamada de nuevo, produciendo la expresión del argumento, así como al texto que permanece. A su vez, este texto puede contener más argumentos o puede ser el paréntesis de cierre que finaliza la lista de argumentos.

Esta es la primera parte del analizador:

```
function analizarExpresion(programa) {
  programa = saltarEspacio(programa);
  let emparejamiento, expresion;
  if (emparejamiento = /^"([^"]*)"/.exec(programa)) {
    expresion = {tipo: "valor", valor: emparejamiento[1]};
  } else if (emparejamiento = /^\d+\b/.exec(programa)) {
    expresion = {tipo: "valor", valor: Number(emparejamiento[0])};
```

```
  } else if (emparejamiento = /^[^\s(),"]+/.exec(programa)) {
    expresion = {tipo: "palabra", nombre: emparejamiento[0]};
  } else {
    throw new SyntaxError("Sintaxis inesperada: " + programa);
  }

  return aplicarAnalisis(expresion, programa.slice(emparejamiento
     [0].length));
}

function saltarEspacio(string) {
  let primero = string.search(/\S/);
  if (primero == -1) return "";
  return string.slice(primero);
}
```

Dado que Egg, al igual que JavaScript, permite cualquier cantidad de espacios en blanco entre sus elementos, tenemos que remover repetidamente los espacios en blanco del inicio del string del programa. Para esto nos ayuda la función `saltarEspacio`.

Después de saltar cualquier espacio en blanco, `analizarExpresion` usa tres expresiones regulares para detectar los tres elementos atómicos que Egg soporta: strings, números y palabras. El analizador construye un tipo diferente de estructura de datos dependiendo de cuál coincida. Si la entrada no coincide con ninguna de estas tres formas, la expresión no es válida, y el analizador arroja un error. Usamos `SyntaxError` en lugar de `Error` como constructor de excepción, el cual es otro tipo de error estándar, dado que es un poco más específico—también es el tipo de error lanzado cuando se intenta ejecutar un programa de JavaScript no válido.

Luego cortamos la parte que coincidio del string del programa y pasamos eso, junto con el objeto para la expresión, a `aplicarAnalisis`, el cual verifica si la expresión es una aplicación. Si es así, analiza una lista de los argumentos entre paréntesis.

```
function aplicarAnalisis(expresion, programa) {
  programa = saltarEspacio(programa);
  if (programa[0] != "(") {
    return {expresion: expresion, resto: programa};
  }

  programa = saltarEspacio(programa.slice(1));
  expresion = {tipo: "aplicar", operador: expresion, argumentos:
```

```
      []};
  while (programa[0] != ")") {
    let argumento = analizarExpresion(programa);
    expresion.argumentos.push(argumento.expresion);
    programa = saltarEspacio(argumento.resto);
    if (programa[0] == ",") {
      programa = saltarEspacio(programa.slice(1));
    } else if (programa[0] != ")") {
      throw new SyntaxError("Experaba ',' o ')'");
    }
  }
  return aplicarAnalisis(expresion, programa.slice(1));
}
```

Si el siguiente carácter en el programa no es un paréntesis de apertura, esto no es una aplicación, y `aplicarAnalisis` retorna la expresión que se le dio.

De lo contrario, salta el paréntesis de apertura y crea el objeto de árbol de sintaxis para esta expresión de aplicación. Entonces, recursivamente llama a `analizarExpresion` para analizar cada argumento hasta que se encuentre el paréntesis de cierre. La recursión es indirecta, a través de `aplicarAnalisis` y `analizarExpresion` llamando una a la otra.

Dado que una expresión de aplicación puede ser aplicada a sí misma (como en `multiplicador(2)(1)`), `aplicarAnalisis` debe, después de haber analizado una aplicación, llamarse asi misma de nuevo para verificar si otro par de paréntesis sigue a continuación.

Esto es todo lo que necesitamos para analizar Egg. Envolvemos esto en una conveniente función `analizar` que verifica que ha llegado al final del string de entrada después de analizar la expresión (un programa Egg es una sola expresión), y eso nos da la estructura de datos del programa.

```
function analizar(programa) {
  let {expresion, resto} = analizarExpresion(programa);
  if (saltarEspacio(resto).length > 0) {
    throw new SyntaxError("Texto inesperado despues de programa");
  }
  return expresion;
}

console.log(analizar("+(a, 10)"));
// → {tipo: "aplicar",
//    operador: {tipo: "palabra", nombre: "+"},
//    argumentos: [{tipo: "palabra", nombre: "a"},
//           {tipo: "valor", valor: 10}]}
```

Funciona! No nos da información muy útil cuando falla y no almacena la línea y la columna en que comienza cada expresión, lo que podría ser útil al informar errores más tarde, pero es lo suficientemente bueno para nuestros propósitos.

The evaluator

What can we do with the syntax tree for a program? Run it, of course! And that is what the evaluator does. You give it a syntax tree and a scope object that associates names with values, and it will evaluate the expression that the tree represents and return the value that this produces.

```
const specialForms = Object.create(null);

function evaluate(expresion, scope) {
  if (expresion.type == "value") {
    return expresion.value;
  } else if (expresion.type == "word") {
    if (expresion.name in scope) {
      return scope[expresion.name];
    } else {
      throw new ReferenceError(
        `Undefined binding: ${expresion.name}`);
    }
  } else if (expresion.type == "apply") {
    let {operator, args} = expresion;
    if (operator.type == "word" &&
        operator.name in specialForms) {
      return specialForms[operator.name](expresion.args, scope);
    } else {
      let op = evaluate(operator, scope);
      if (typeof op == "function") {
        return op(...args.map(arg => evaluate(arg, scope)));
      } else {
        throw new TypeError("Applying a non-function.");
      }
    }
  }
}
```

The evaluator has code for each of the expression types. A literal value expression produces its value. (For example, the expression `100` just evaluates

to the number 100.) For a binding, we must check whether it is actually defined in the scope and, if it is, fetch the binding's value.

Applications are more involved. If they are a special form, like `if`, we do not evaluate anything and pass the argument expressions, along with the scope, to the function that handles this form. If it is a normal call, we evaluate the operator, verify that it is a function, and call it with the evaluated arguments.

We use plain JavaScript function values to represent Egg's function values. We will come back to this later, when the special form called `fun` is defined.

The recursive structure of `evaluate` resembles the similar structure of the parser, and both mirror the structure of the language itself. It would also be possible to integrate the parser with the evaluator and evaluate during parsing, but splitting them up this way makes the program clearer.

This is really all that is needed to interpret Egg. It is that simple. But without defining a few special forms and adding some useful values to the environment, you can't do much with this language yet.

Special forms

The `specialForms` object is used to define special syntax in Egg. It associates words with functions that evaluate such forms. It is currently empty. Let's add `if`.

```
specialForms.si = (args, scope) => {
  if (args.length != 3) {
    throw new SyntaxError("Wrong number of args to si");
  } else if (evaluate(args[0], scope) !== false) {
    return evaluate(args[1], scope);
  } else {
    return evaluate(args[2], scope);
  }
};
```

Egg's `si` construct expects exactly three arguments. It will evaluate the first, and if the result isn't the value `false`, it will evaluate the second. Otherwise, the third gets evaluated. This `if` form is more similar to JavaScript's ternary `?:` operator than to JavaScript's `if`. It is an expression, not a statement, and it produces a value, namely the result of the second or third argument.

Egg also differs from JavaScript in how it handles the condition value to `if`. It will not treat things like zero or the empty string as false, only the precise value `false`.

The reason we need to represent `if` as a special form, rather than a regular function, is that all arguments to functions are evaluated before the function is called, whereas `if` should evaluate only *either* its second or its third argument, depending on the value of the first.

The `while` form is similar.

```
specialForms.while = (args, scope) => {
  if (args.length != 2) {
    throw new SyntaxError("Wrong number of args to while");
  }
  while (evaluate(args[0], scope) !== false) {
    evaluate(args[1], scope);
  }

  // Since undefined does not exist in Egg, we return false,
  // for lack of a meaningful result.
  return false;
};
```

Another basic building block is `hacer`, which executes all its arguments from top to bottom. Its value is the value produced by the last argument.

```
specialForms.hacer = (args, scope) => {
  let value = false;
  for (let arg of args) {
    value = evaluate(arg, scope);
  }
  return value;
};
```

To be able to create bindings and give them new values, we also create a form called `definir`. It expects a word as its first argument and an expression producing the value to assign to that word as its second argument. Since `definir`, like everything, is an expression, it must return a value. We'll make it return the value that was assigned (just like JavaScript's `=` operator).

```
specialForms.definir = (args, scope) => {
  if (args.length != 2 || args[0].type != "word") {
    throw new SyntaxError("Incorrect use of definir");
  }
  let value = evaluate(args[1], scope);
  scope[args[0].name] = value;
  return value;
```

```
};
```

The environment

The scope accepted by `evaluate` is an object with properties whose names correspond to binding names and whose values correspond to the values those bindings are bound to. Let's define an object to represent the global scope.

To be able to use the `if` construct we just defined, we must have access to Boolean values. Since there are only two Boolean values, we do not need special syntax for them. We simply bind two names to the values `true` and `false` and use those.

```
const topScope = Object.create(null);

topScope.true = true;
topScope.false = false;
```

We can now evaluate a simple expression that negates a Boolean value.

```
let prog = parse(`si(true, false, true)`);
console.log(evaluate(prog, topScope));
// → false
```

To supply basic arithmetic and comparison operators, we will also add some function values to the scope. In the interest of keeping the code short, we'll use `Function` to synthesize a bunch of operator functions in a loop, instead of defining them individually.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
  topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

A way to output values is also very useful, so we'll wrap `console.log` in a function and call it `imprimir`.

```
topScope.imprimir = value => {
  console.log(value);
  return value;
};
```

That gives us enough elementary tools to write simple programs. The following function provides a convenient way to parse a program and run it in a fresh scope.

```
function run(programa) {
  return evaluate(parse(programa), Object.create(topScope));
}
```

We'll use object prototype chains to represent nested scopes, so that the program can add bindings to its local scope without changing the top-level scope.

```
run(`
hacer(definir(total, 0),
   definir(count, 1),
   while(<(count, 11),
         hacer(definir(total, +(total, count)),
            definir(count, +(count, 1)))),
   imprimir(total))
`);
// → 55
```

This is the program we've seen several times before, which computes the sum of the numbers 1 to 10, expressed in Egg. It is clearly uglier than the equivalent JavaScript program—but not bad for a language implemented in less than 150 lines of code.

Functions

A programming language without functions is a poor programming language indeed.

Fortunately, it isn't hard to add a `fun` construct, which treats its last argument as the function's body and uses all arguments before that as the names of the function's parameters.

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Functions need a body");
  }
  let body = args[args.length - 1];
  let params = args.slice(0, args.length - 1).map(expr => {
```

```
    if (expr.type != "word") {
      throw new SyntaxError("Parameter names must be words");
    }
    return expr.name;
  });

  return function() {
    if (arguments.length != params.length) {
      throw new TypeError("Wrong number of arguments");
    }
    let localScope = Object.create(scope);
    for (let i = 0; i < arguments.length; i++) {
      localScope[params[i]] = arguments[i];
    }
    return evaluate(body, localScope);
  };
};
```

Functions in Egg get their own local scope. The function produced by the `fun` form creates this local scope and adds the argument bindings to it. It then evaluates the function body in this scope and returns the result.

```
run(`
hacer(definir(plusOne, fun(a, +(a, 1))),
   imprimir(plusOne(10)))
`);
// → 11

run(`
hacer(definir(pow, fun(base, exp,
     si(==(exp, 0),
        1,
        *(base, pow(base, -(exp, 1))))))),
   imprimir(pow(2, 10)))
`);
// → 1024
```

Compilation

What we have built is an interpreter. During evaluation, it acts directly on the representation of the program produced by the parser.

Compilation is the process of adding another step between the parsing and

the running of a program, which transforms the program into something that can be evaluated more efficiently by doing as much work as possible in advance. For example, in well-designed languages it is obvious, for each use of a binding, which binding is being referred to, without actually running the program. This can be used to avoid looking up the binding by name every time it is accessed, instead directly fetching it from some predetermined memory location.

Traditionally, compilation involves converting the program to machine code, the raw format that a computer's processor can execute. But any process that converts a program to a different representation can be thought of as compilation.

It would be possible to write an alternative evaluation strategy for Egg, one that first converts the program to a JavaScript program, uses `Function` to invoke the JavaScript compiler on it, and then runs the result. When done right, this would make Egg run very fast while still being quite simple to implement.

If you are interested in this topic and willing to spend some time on it, I encourage you to try to implement such a compiler as an exercise.

Cheating

When we defined `if` and `while`, you probably noticed that they were more or less trivial wrappers around JavaScript's own `if` and `while`. Similarly, the values in Egg are just regular old JavaScript values.

If you compare the implementation of Egg, built on top of JavaScript, with the amount of work and complexity required to build a programming language directly on the raw functionality provided by a machine, the difference is huge. Regardless, this example hopefully gave you an impression of the way programming languages work.

And when it comes to getting something done, cheating is more effective than doing everything yourself. Though the toy language in this chapter doesn't do anything that couldn't be done better in JavaScript, there *are* situations where writing small languages helps get real work done.

Such a language does not have to resemble a typical programming language. If JavaScript didn't come equipped with regular expressions, for example, you could write your own parser and evaluator for regular expressions.

Or imagine you are building a giant robotic dinosaur and need to program its behavior. JavaScript might not be the most effective way to do this. You might instead opt for a language that looks like this:

```
behavior walk
```

```
  perform when
    destination ahead
  actions
    move left-foot
    move right-foot

behavior attack
  perform when
    Godzilla in-view
  actions
    fire laser-eyes
    launch arm-rockets
```

This is what is usually called a *domain-specific language*, a language tailored to express a narrow domain of knowledge. Such a language can be more expressive than a general-purpose language because it is designed to describe exactly the things that need to be described in its domain, and nothing else.

Exercises

Arrays

Add support for arrays to Egg by adding the following three functions to the top scope: `array(...values)` to construct an array containing the argument values, `length(array)` to get an array's length, and `element(array, n)` to fetch the n^{th} element from an array.

Closure

The way we have defined `fun` allows functions in Egg to reference the surrounding scope, allowing the function's body to use local values that were visible at the time the function was defined, just like JavaScript functions do.

The following program illustrates this: function `f` returns a function that adds its argument to `f`'s argument, meaning that it needs access to the local scope inside `f` to be able to use binding `a`.

```
run(`
hacer(definir(f, fun(a, fun(b, +(a, b)))),
   imprimir(f(4)(5)))
`);
// → 9
```

Go back to the definition of the `fun` form and explain which mechanism causes this to work.

Comments

It would be nice if we could write comments in Egg. For example, whenever we find a hash sign (`#`), we could treat the rest of the line as a comment and ignore it, similar to `//` in JavaScript.

We do not have to make any big changes to the parser to support this. We can simply change `skipSpace` to skip comments as if they are whitespace so that all the points where `skipSpace` is called will now also skip comments. Make this change.

Fixing scope

Currently, the only way to assign a binding a value is `definir`. This construct acts as a way both to define new bindings and to give existing ones a new value.

This ambiguity causes a problem. When you try to give a nonlocal binding a new value, you will end up defining a local one with the same name instead. Some languages work like this by design, but I've always found it an awkward way to handle scope.

Add a special form `set`, similar to `definir`, which gives a binding a new value, updating the binding in an outer scope if it doesn't already exist in the inner scope. If the binding is not defined at all, throw a `ReferenceError` (another standard error type).

The technique of representing scopes as simple objects, which has made things convenient so far, will get in your way a little at this point. You might want to use the `Object.getPrototypeOf` function, which returns the prototype of an object. Also remember that scopes do not derive from `Object.prototype`, so if you want to call `hasOwnProperty` on them, you have to use this clumsy expression:

```
Object.prototype.hasOwnProperty.call(scope, name);
```

"¡Tanto peor! ¡Otra vez la vieja historia! Cuando uno ha acabado de construir su casa advierte que, mientras la construía, ha aprendido, sin darse cuenta, algo que tendría que haber sabido absolutamente antes de comenzar a construir."

—Friedrich Nietzsche, Más allá del bien y del mal

Chapter 13

El Modelo de Objeto del Documento

Cuando abres una página web en tu navegador, el navegador obtiene el texto de la página HTML y lo analiza, de una manera bastante similar a la manera en que nuestro analizador del Capítulo 12 analizaba los programas. El navegador construye un modelo de la estructura del documento y utiliza este modelo para dibujar la página en la pantalla.

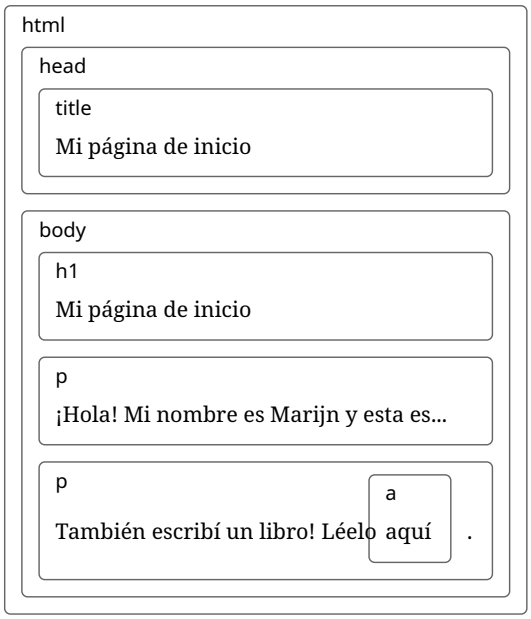
Esta representación del documento es uno de los juguetes que un programa de JavaScript tiene disponible en su caja de arena. Es una estructura de datos que puedes leer o modificar. Y actúa como una estructura en *tiempo real*: cuando se modifica, la página en la pantalla es actualizada para reflejar los cambios.

La estructura del documento

Te puedes imaginar a un documento HTML como un conjunto anidado de cajas. Las etiquetas como `<body>` y `</body>` encierran otras etiquetas, que a su vez, contienen otras etiquetas o texto. Este es el documento de ejemplo del capítulo anterior:

```
<!doctype html>
<html>
  <head>
    <title>Mi página de inicio</title>
  </head>
  <body>
    <h1>Mi página de inicio</h1>
    <p>Hola, mi nombre es Marijn y esta es mi página de inicio.</p>
    <p>También escribí un libro! Léelo
      <a href="http://eloquentjavascript.net">aquí</a>.</p>
  </body>
</html>
```

Esta página tiene la siguiente estructura:



La estructura de datos que el navegador utiliza para representar el documento sigue esta figura. Para cada caja, hay un objeto, con el que podemos interactuar para descubrir cosas como que etiqueta de HTML representa y que cajas y texto contiene. Esta representación es llamada *Modelo de Objeto del Documento* o DOM *(por sus siglas en inglés "Document Object Model")*.

El objeto de enlace global `document` nos da acceso a esos objetos. Su propiedad `documentElement` hace referencia al objeto que representa a la etiqueta `<html>`. Dado que cada documento HTML tiene una cabecera y un cuerpo, también tiene propiedades `head` y `body` que apuntan a esos elementos.

Árboles

Pensemos en los árboles sintácticos del Capítulo 12 por un momento. Sus estructuras son sorprendentemente similares a la estructura de un documento del navegador. Cada *nodo* puede referirse a otros nodos *hijos* que, a su vez, pueden tener hijos propios. Esta forma es típica de las estructuras anidadas donde los elementos pueden contener sub elementos que son similares a ellos mismos.

Le damos el nombre de *árbol* a una estructura de datos cuando tiene una estructura de ramificación, no tiene ciclos (un nodo no puede contenerse a sí mismo, directa o indirectamente) y tiene una única *raíz* bien definida. En el caso del *DOM*, `document.documentElement` hace la función de raíz.

Los árboles aparecen constantemente en las ciencias de la computación *(computer sience)*. Además de representar estructuras recursivas como los docu-

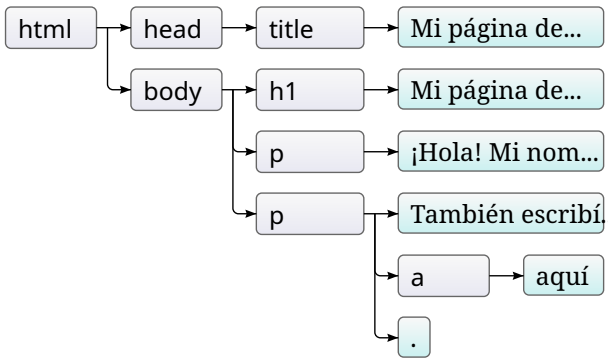
mentos HTML o programas, también son comúnmente usados para mantener conjuntos ordenados de datos debido a que los elementos generalmente pueden ser encontrados o agregados más eficientemente en un árbol que en un arreglo plano.

Un árbol típico tiene diferentes tipos de nodos. El árbol sintáctico del lenguaje *Egg* tenía identificadores, valores y nodos de aplicación. Los nodos de aplicación pueden tener hijos, mientras que los identificadores y valores son *hojas*, o nodos sin hijos.

Lo mismo sucede para el DOM, los nodos para los *elementos*, los cuales representan etiquetas HTML, determinan la estructura del documento. Estos pueden tener nodos hijos. Un ejemplo de estos nodos es `document.body`. Algunos de estos hijos pueden ser nodos hoja, como los fragmentos de texto o los nodos comentario.

Cada objeto *nodo DOM* tiene una propiedad `nodeType`, la cual contiene un código (numérico) que identifica el tipo de nodo. Los *Elementos* tienen el código 1, que también es definido por la propiedad constante `Node.ELEMENT_NODE`. Los nodos de texto, representando una sección de texto en el documento, obtienen el código 3 (`Node.TEXT_NODE`). Los comentarios obtienen el código 8 (`Node.COMMENT_NODE`).

Otra forma de visualizar nuestro árbol de documento es la siguiente:



Las hojas son nodos de texto, y las flechas nos indican las relaciones padre-hijo entre los nodos.

El estándar

Usar códigos numéricos crípticos para representar a los tipos de nodos no es algo que se parezca al estilo de JavaScript para hacer las cosas. Más adelante en este capítulo, veremos cómo otras partes de la interfaz DOM también se sienten engorrosas y alienígenas. La razón de esto es que DOM no fue diseñado

solamente para JavaScript. Más bien, intenta ser una interfaz *independiente del lenguaje* que puede ser usada en otros sistemas, no solamente para HTML pero también para XML, que es un formato de datos genérico con una sintaxis similar a la de HTML.

Esto es desafortunado. Usualmente los estándares son bastante útiles. Pero en este caso, la ventaja (consistencia entre lenguajes) no es tan conveniente. Tener una interfaz que está propiamente integrada con el lenguaje que estás utilizando te ahorrará más tiempo que tener una interfaz familiar en distintos lenguajes.

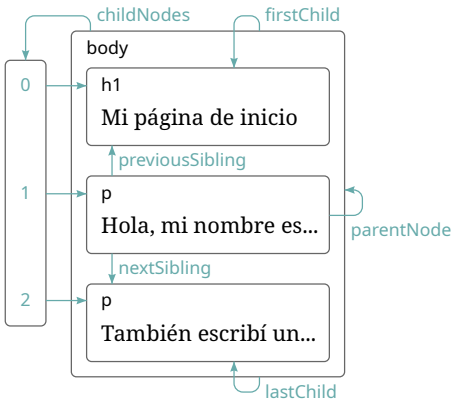
A manera de ejemplo de esta pobre integración, considera la propiedad `childNodes` que los nodos elemento en el DOM tienen. Esta propiedad almacena un objeto parecido a un arreglo, con una propiedad `length` y propiedades etiquetadas por números para acceder a los nodos hijos. Pero es una instancia de tipo `NodeList`, no un arreglo real, por lo que no tiene métodos como `slice` o `map`.

Luego, hay problemas que son simplemente un pobre diseño. Por ejemplo, no hay una manera de crear un nuevo nodo e inmediatamente agregar hijos o attributos. En vez de eso, tienes que crearlo primero y luego agregar los hijos y atributos uno por uno, usando efectos secundarios. El código que interactúa mucho con el DOM tiende a ser largo, repetitivo y feo.

Pero estos defectos no son fatales. Dado que JavaScript nos permite crear nuestra propias abstracciones, es posible diseñar formas mejoradas para expresar las operaciones que estás realizando. Muchas bibliotecas destinadas a la programación del navegador vienen con esas herramientas.

Moviéndose a través del árbol

Los nodos del DOM contienen una amplia cantidad de enlaces a otros nodos cercanos. El siguiente diagrama los ilustra:



A pesar de que el diagrama muestra un solo enlace por cada tipo, cada nodo tiene una propiedad `parentNode` que apunta al nodo al que pertenece, si es que hay alguno. Igualmente, cada nodo *elemento* (nodo tipo 1) tiene una propiedad `childNodes` que apunta a un objeto similar a un arreglo que almacena a sus hijos.

En teoría, te deberías poder mover donde quieras en el árbol utilizando únicamente estos enlaces entre padre e hijo. Pero JavaScript también te otorga acceso a un número de enlaces adicionales convenientes. Las propiedades `firstChild` y `lastChild` apuntan al primer y último elementos hijo, o tiene el valor `null` para nodos sin hijos. De manera similar, las propiedades `previousSibling` y `nextSibling` apuntan a los nodos adyacentes, los cuales, son nodos con el mismo padre que aparecen inmediatamente antes o después del nodo. Para el primer hijo `previousSibling` será `null` y para el último hijo, `nextSibling` será `null`.

También existe una propiedad `children`, que es parecida a `childNodes` pero contiene únicamente hijos de tipo *elemento* (tipo 1), excluyendo otros tipos de nodos. Esto puede ser útil cuando no estás interesando en nodos de tipo texto.

Cuando estás tratando con estructuras de datos anidadas como esta, las funciones recursivas son generalmente útiles. La siguiente función escanea un documento por nodos de texto que contengan una cadena dada y regresan `true` en caso de que encuentren una:

```
function hablaSobre(nodo, cadena) {
  if (nodo.nodeType == Node.ELEMENT_NODE) {
    for (let i = 0; i < nodo.childNodes.length; i++) {
      if (hablaSobre(nodo.childNodes[i], cadena)) {
        return true;
      }
    }
    return false;
  } else if (nodo.nodeType == Node.TEXT_NODE) {
    return nodo.nodeValue.indexOf(cadena) > -1;
  }
}

console.log(hablaSobre(document.body, "libro"));
// → true
```

Debido a que `childNodes` no es un arreglo real, no podemos iterar sobre el usando `for/of` por lo que tenemos que iterar sobre el rango del índice usando un `for` regular o usando `Array.from`.

La propiedad `nodeValue` de un nodo de texto almacena la cadena de texto que representa.

Buscar elementos

Navegar por estos enlaces entre padres, hijos y hermanos suele ser útil. Pero si queremos encontrar un nodo específico en el documento, alcanzarlo comenzando en `document.body` y siguiendo un camino fijo de propiedades es una mala idea. Hacerlo genera suposiciones en nuestro programa sobre la estructura precisa del documento que tal vez quieras cambiar después. Otro factor complicado es que los nodos de texto son creados incluso para los espacios en blanco entre nodos. La etiqueta `<body>` en el documento de ejemplo no tiene solamente tres hijos (`<h1>` y dos elementos `<p>`), en realidad tiene siete: esos tres, más los espacios posteriores y anteriores entre ellos.

Por lo que si queremos obtener el atributo `href` del enlace en ese documento, no queremos decir algo como "Obten el segundo hijo del sexto hijo del elemento *body* del documento". Sería mejor si pudiéramos decir "Obten el primer enlace en el documento". Y de hecho podemos.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

Todos los nodos *elemento* tienen un método `getElementsByTagName`, el cual recolecta a todos los elementos con un nombre de etiqueta dado que son descendientes (hijos directos o indirectos) de ese nodo y los regresa como un objeto parecido a un arreglo.

Para encontrar un *único* nodo en específico, puedes otorgarle un atributo `id` y usar `document.getElementById`.

```
<p>Mi avestruz Gertrudiz:</p>
<p><img id="gertrudiz" src="img/ostrich.png"></p>

<script>
  let avestruz = document.getElementById("gertrudiz");
  console.log(avestruz.src);
</script>
```

Un tercer método similar es `getElementsByClassName`, el cual, de manera similar a `getElementsByTagName` busca a través de los contenidos de un nodo elemento y obtiene todos los elementos que tienen una cadena dada en su

attributo `class`.

Actualizar el documento

Prácticamente todo sobre la estructura de datos DOM puede ser cambiado. La forma del árbol de documento puede ser modificada cambiando las relaciones padre-hijo. Los nodos tienen un método `remove` para ser removidos de su nodo padre actual. Para agregar un nodo hijo a un nodo elemento, podemos usar `appendChild`, que lo pondrá al final de la lista de hijos, o `insertBefore`, que insertará el nodo en el primer argumento antes del nodo en el segundo argumento.

```
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>

<script>
  let parrafos = document.body.getElementsByTagName("p");
  document.body.insertBefore(parrafos[2], parrafos[0]);
</script>
```

Un nodo puede existir en el documento solamente en un lugar. En consecuencia, insertar el párrafo *Tres* enfrente del párrafo *Uno* primero lo removerá del final del documento y luego lo insertará en la parte delantera, resultando en *Tres/Uno/Dos*. Todas las operaciones que insertan un nodo en alguna parte causarán, a modo de efecto secundario, que el nodo sea removido de su posición actual (si es que tiene una).

El método `replaceChild` es usado para reemplazar a un nodo hijo con otro. Toma dos nodos como argumentos: un nuevo nodo y el nodo que será reemplazado. El nodo reemplazado debe ser un nodo hijo del elemento desde donde se está llamando el método. Nótese que tanto `replaceChild` como `insertBefore` esperan que el *nuevo* nodo sea el primer argumento.

Crear nodos

Digamos que queremos escribir un *script* que reemplace todas las imagenes (etiquetas `<img>`) en el documento con el texto contenido en sus atributos `alt`, los cuales especifican una representación textual alternativa de la imagen.

Esto no solamente involucra remover las imágenes, si no que también involucra agregar un nuevo nodo texto que las reemplace. Los nodos texto son

creados con el método `document.createTextNode`.

```
<p>El <img src="img/cat.png" alt="Gato"> en el
  <img src="img/hat.png" alt="Sombrero">.</p>

<p><button onclick="sustituirImagenes()">Sustituir</button></p>

<script>
  function sustituirImagenes() {
    let imagenes = document.body.getElementsByTagName("img");
    for (let i = imagenes.length - 1; i >= 0; i--) {
      let imagen = imagenes[i];
      if (imagen.alt) {
        let texto = document.createTextNode(imagen.alt);
        imagen.parentNode.replaceChild(texto, imagen);
      }
    }
  }
</script>
```

Dada una cadena, `createTextNode` nos da un nodo texto que podemos insertar en el documento para hacer que aparezca en la pantalla.

El ciclo que recorre las imágenes empieza al final de la lista. Esto es necesario dado que la lista de nodos regresada por un método como `getElementsByTagName` (o una propiedad como `childNodes`) se actualiza en tiempo real. Esto es, que se actualiza conforme el documento cambia. Si empezáramos desde el frente, remover la primer imagen causaría que la lista perdiera su primer elemento de tal manera que la segunda ocasión que el ciclo se repitiera, donde `i` es 1, se detendría dado que la longitud de la colección ahora es también 1.

Si quieres una colección de nodos *sólida*, a diferencia de una en tiempo real, puedes convertir la colección a un arreglo real llamando `Array.from`.

```
let casi_arreglo = {0: "uno", 1: "dos", length: 2};
let arreglo = Array.from(casi_arreglo);
console.log(arreglo.map(s => s.toUpperCase()));
// → ["UNO", "DOS"]
```

Para crear nodos elemento, puedes utilizar el método `document.createElement`. Este método toma un nombre de etiqueta y regresa un nuevo nodo vacío del tipo dado.

El siguiente ejemplo define una utilidad `elt`, la cual crea un elemento nodo y trata el resto de sus argumentos como hijos de ese nodo. Luego, esta función

es utilizada para agregar una atribución a una cita.

```
<blockquote id="cita">
  Ningún libro puede terminarse jamás. Mientras se trabaja en
  él aprendemos solo lo suficiente para encontrar inmaduro
  el momento en el que nos alejamos de él.
</blockquote>

<script>
  function elt(tipo, ...hijos) {
    let nodo = document.createElement(tipo);
    for (let hijo of hijos) {
      if (typeof hijo != "string") nodo.appendChild(hijo);
      else nodo.appendChild(document.createTextNode(hijo));
    }
    return nodo;
  }

  document.getElementById("cita").appendChild(
    elt("footer", "—",
        elt("strong", "Karl Popper"),
        ", prefacio de la segunda edición de ",
        elt("em", "La sociedad abierta y sus enemigos"),
        ", 1950"));
</script>
```

Así es como se ve el documento resultante:

> No book can ever be finished. While working on it we learn just enough to find it immature the moment we turn away from it.
> —**Karl Popper**, preface to the second editon of *The Open Society and Its Enemies*, 1950

Atributos

Los atributos de algunos elementos, como `href` para los enlaces, pueden ser accedidos a través de una propiedad con el mismo nombre en el objeto DOM del elemento. Este es el caso para los atributos estándar más comúnmente utilizados.

Pero HTML te permite establecer cualquier atributo que quieras en los nodos. Esto puede ser útil debido a que te permite almacenar información extra en un documento. Sin embargo, si creas tus propios nombres de atributo, dichos atributos no estarán presentes como propiedades en el nodo del elemento. En

vez de eso, tendrás que utilizar los métodos `getAttribute` y `setAttribute` para poder trabajar con ellos.

```
<p data-classified="secreto">El código de lanzamiento es:
   00000000.</p>
<p data-classified="no-classificado">Yo tengo dos pies.</p>

<script>
  let parrafos = document.body.getElementsByTagName("p");
  for (let parrafo of Array.from(parrafos)) {
    if (parrafo.getAttribute("data-classified") == "secreto") {
      parrafo.remove();
    }
  }
</script>
```

Se recomienda anteponer los nombres de dichos atributos inventados con `data-` para asegurarse de que no conflictúan con ningún otro atributo.

Existe un atributo comúnmente usado, `class`, que es una palabra clave en el lenguaje JavaScript. Por motivos históricos, algunas implementaciones antiguas de JavaScript podrían no manejar nombres de propiedades que coincidan con las palabras clave, la propiedad utilizada para acceder a este atributo tiene por nombre `className`. También puedes acceder a ella bajo su nombre real, `"class"`, utilizando los métodos `getAttribute` y `setAttribute`.

Layout

Tal vez hayas notado que diferentes tipos de elementos se exponen de manera distinta. Algunos, como en el caso de los párrafos (`<p>`) o encabezados (`<h1>`), ocupan todo el ancho del documento y se renderizan en líneas separadas. A estos se les conoce como elementos *block* (o bloque). Otros, como los enlaces (`<a>`) o el elemento `<strong>`, se renderizan en la misma línea con su texto circundante. Dichos elementos se les conoce como elementos *inline* (o en línea).

Para cualquier documento dado, los navegadores son capaces de calcular una estructura *(layout)*, que le da a cada elemento un tamaño y una posición basada en el tipo y el contenido. Luego, esta estructura se utiliza para trazar el documento.

Se puede acceder al tamaño y la posición de un elemento desde JavaScript. Las propiedades `offsetWidth` y `offsetHeight` te dan el espacio que el elemento utiliza en pixeles. Un píxel es la unidad básica de las medidas del navegador. Tradicionalmente correspondía al punto más pequeño que la pantalla podía

trazar, pero en los monitores modernos, que pueden trazar puntos *muy* pequeños, este puede no ser más el caso, por lo que un píxel del navegador puede abarcar varios puntos en la pantalla.

De manera similar, `clientWidth` y `clientHeight` te dan el tamaño del espacio *dentro* del elemento, ignorando la anchura del borde.

```
<p style="border: 3px solid red">
  Estoy dentro de una caja
</p>

<script>
  let parrafo = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", parrafo.clientHeight);
  console.log("offsetHeight:", parrafo.offsetHeight);
</script>
```

Darle un borde a un párrafo hace que se dibuje un rectángulo a su alrededor.

La manera más efectiva de encontrar la posición precisa de un elemento en la pantalla es el método `getBoundingClientRect`. Este devuelve un objeto con las propiedades `top`, `bottom`, `left`, y `right`, indicando las posiciones de pixeles de los lados el elemento en relación con la parte superior izquierda de la pantalla. Si los quieres en relación a todo el documento, deberás agregar la posición actual del scroll, la cual puedes obtener en los *bindings* `pageXOffset` y `pageYOffset`.

Estructurar un documento puede requerir mucho trabajo. En los intereses de velocidad, los motores de los navegadores no reestructuran inmediatamente un documento cada vez que lo cambias, en cambio, se espera lo más que se pueda. Cuando un programa de JavaScript que modifica el documento termina de ejecutarse, el navegador tendrá que calcular una nueva estructura para trazar el documento actualizado en la pantalla. Cuando un programa *solicita* la posición o el tamaño de algo, leyendo propiedades como `offsetHeight` o llamando a `getBoundingClientRect`, proveer la información correcta también requiere que se calcule una nueva estructura.

A un programa que alterna repetidamente entre leer la información de la estructura DOM y cambiar el DOM, fuerza a que haya bastantes cálculos de estructura, y por consecuencia se ejecutará lentamente. El siguiente código es un ejemplo de esto. Contiene dos programas diferentes que construyen una línea de X caracteres con 2,000 pixeles de ancho y que mide el tiempo que toma

cada uno.

```
<p><span id="uno"></span></p>
<p><span id="dos"></span></p>

<script>
  function tiempo(nombre, accion) {
    let inicio = Date.now(); // Tiempo actual en milisegundos
    accion();
    console.log(nombre, "utilizo", Date.now() - inicio, "ms");
  }

  tiempo("inocente", () => {
    let objetivo = document.getElementById("uno");
    while (objetivo.offsetWidth < 2000) {
      objetivo.appendChild(document.createTextNode("X"));
    }
  });
  // → inocente utilizo 32 ms

  tiempo("ingenioso", function() {
    let objetivo = document.getElementById("dos");
    objetivo.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (objetivo.offsetWidth / 5));
    objetivo.firstChild.nodeValue = "X".repeat(total);
  });
  // → ingenioso utilizo 1 ms
</script>
```

Estilización

Hemos visto que diferentes elementos HTML se trazan de manera diferente. Algunos son desplegados como bloques, otros en línea. Algunos agregan estilos, por ejemplo `<strong>` hace que su contenido esté en negritas y `<a>` lo hace azul y lo subraya.

La forma en la que una etiqueta `<img>` muestra una imagen o una etiqueta `<a>` hace que un enlace sea seguido cuando se hace click en el, está fuertemente atado al tipo del elemento. Pero podemos cambiar los estilos asociados a un elemento, tales como el color o si está subrayado. Este es un ejemplo que utiliza la propiedad `style`:

```
<p><a href=".">Enlace normal</a></p>
```

```
<p><a href="." style="color: green">Enlace verde</a></p>
```

El segundo enlace será verde en vez del color por defecto.

Normal link

Green link

Un atributo *style* puede llegar a contener una o más declaraciones, que consisten en una propiedad (como `color`) seguido del símbolo de dos puntos y un valor (como `green`). Cuando hay más de una declaración, estas deben ser separadas por punto y coma, como en `"color: red; border: none"`.

Muchos de los aspectos del documento pueden ser influenciados por la estilización. Por ejemplo, la propiedad `display` controla si un elemento es desplegado como un bloque o como un elemento en línea.

```
El texto es desplegado <strong>en línea</strong>,
<strong style="display: block">como bloque</strong>, y
<strong style="display: none">no se despliega</strong>.
```

La etiqueta `block` terminará en su propia línea dado que los elementos *bloque* no son desplegados en línea con el texto que los rodea. La última etiqueta no se despliega—`display: none` previene que el elemento sea mostrado en la pantalla. Esta es una manera de ocultar elementos. A menudo esto es preferido sobre removerlos completamente del documento debido a que hace más fácil mostrarlos nuevamente en el futuro.

This text is displayed **inline**,
as a block
, and .

El código de JavaScript puede manipular directamente el estilo de un elemento a través de la propiedad `style` del elemento. Esta propiedad almacena un objeto que tiene propiedades por todas las posibles propiedades de estilo. Los valores de esas propiedades son cadenas, que podemos escribir para cambiar un aspecto en particular del estilo del elemento.

```
<p id="parrafo" style="color: purple">
  Texto mejorado
</p>

<script>
  let parrafo = document.getElementById("parrafo");
```

```
  console.log(parrafo.style.color);
  parrafo.style.color = "magenta";
</script>
```

Algunos nombres de propiedades pueden contener guiones, como es el caso de `font-family`. Dado que estos nombres de propiedades son incómodos para trabajar con ellos en JavaScript (tendrías que decir `style["font-family"]`), los nombres de propiedades en el objeto `style` para tales propiedades no tendrán guiones y las letra después del guion estará en mayúsculas (`style.fontFamily`).

Estilos en Cascada

El sistema de estilos para HTML es llamado CSS por sus siglas en ingles *Cascading Style Sheets* (Hojas de estilo en cascada). Una *hoja de estilos* es un conjunto de reglas sobre cómo estilizar a los elementos en un documento. Puede estar declarado dentro de una etiqueta `<style>`.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Ahora <strong>el texto en negritas</strong> esta en italicas y es
    gris.</p>
```

La sección *cascada* en el nombre se refiere al hecho de que varias reglas son combinadas para producir el estilo final para un elemento. En el ejemplo, el estilo por defecto para las etiquetas `<strong>`, que les da `font-weight: bold`, es superpuesto por la regla en la etiqueta `<style>`, que le agrega `font-style` y `color`.

Cuando varias reglas definen un valor para una misma propiedad, la regla leída más recientemente obtiene una mayor precedencia y gana. Por lo que si la regla en la etiqueta `<style>` incluyera `font-weight: normal`, contradiciendo la regla por defecto de `font-weight`, el texto se vería normal, *no* en negritas. Los estilos en un atributo `style` aplicados directamente al nodo tienen la mayor precedencia y siempre ganan.

Es posible apuntar a otras cosas que no sean nombres de etiqueta en las reglas CSS. Una regla para `.abc` aplica a todos los elementos con `"abc"` en su atributo `class`. Una regla para `#xyz` aplica a todos los elementos con un

atributo `id` con valor `"xyz"` (que debería ser único en el documento).

```
.sutil {
  color: gray;
  font-size: 80%;
}
#cabecera {
  background: blue;
  color: white;
}
/* Elementos p con un id principal y clases a y b */
p#principal.a.b {
  margin-bottom: 20px;
}
```

La regla de precedencia que favorece a las reglas más recientemente definidas aplican solamente cuando las reglas tienen la misma *especificidad*. La especificidad de una regla es una medida de que tan precisamente describe a los elementos que coinciden con la regla, determinado por el número y tipo (etiqueta, clase o ID) de los aspectos del elemento que requiera. Por ejemplo, una regla que apunta a `p.a` es más específica que las reglas que apuntan a `p` o solamente a `.a` y tendrá precedencia sobre ellas.

La notación `p > a …{}` aplica los estilos dados a todas las etiquetas `<a>` que son hijas directas de las etiquetas `<p>`. De manera similar, `p a …{}` aplica a todas las etiquetas `<a>` dentro de etiquetas `<p>`, sin importar que sean hijas directas o indirectas.

Selectores de consulta

No utilizaremos mucho las hojas de estilo en este libro. Entenderlas es útil cuando se programa en el navegador, pero son lo suficientemente complicadas para justificar un libro por separado.

La principal razón por la que introduje la sintaxis de selección—la notación usada en las hojas de estilo para determinar a cuales elementos aplicar un conjunto de estilos—es que podemos utilizar el mismo mini-lenguaje como una manera efectiva de encontrar elementos DOM.

El método `querySelectorAll`, que se encuentra definido tanto en el objeto `document` como en los nodos elemento, toma la cadena de un selector y regresa una `NodeList` que contiene todos los elementos que coinciden con la consulta.

```
<p>And if you go chasing
```

```
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="caracter">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>

<script>
  function contar(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(contar("p"));           // Todos los elementos <p>
  // → 4
  console.log(contar(".animal"));     // Clase animal
  // → 2
  console.log(contar("p .animal"));   // Animales dentro de <p>
  // → 2
  console.log(contar("p > .animal")); // Hijos directos de <p>
  // → 1
</script>
```

A diferencia de métodos como `getElementsByTagName`, el objeto que regresa `querySelectorAll` no es un objeto en tiempo real. No cambiará cuando cambies el documento. Sin embargo, sigue sin ser un arreglo real, por lo que aún necesitas llamar a `Array.from` si lo quieres tratar como uno real.

El método `querySelector` (sin la parte de `All`) trabaja de una manera similar. Este es útil si quieres un único elemento en específico. Regresará únicamente el primer elemento que coincida o *null* en el caso que ningún elemento coincida.

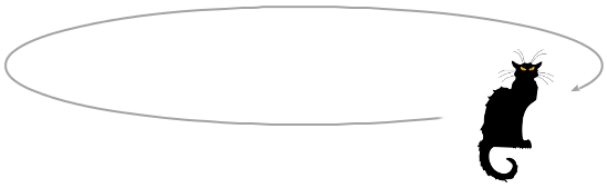
Posicionamiento y animaciones

La propiedad de estilo `position` influye de un manera poderosa sobre la estructura. Por defecto, tiene el valor de `static`, eso significa que el elemento se coloca en su lugar normal en el documento. Cuando se establece como `relative`, el elemento sigue utilizando espacio en el documento pero ahora las propiedades `top` y `left` pueden ser utilizadas para moverlo relativamente a ese espacio normal. Cuando `position` se establece como `absolute`, el elemento es removido del flujo normal del documento—esto es, deja de tomar espacio y puede encimarse con otros elementos. Además, sus propiedades `top` y `left` pueden ser utilizadas para posicionarlo absolutamente con relación a la esquina superior izquierda del elemento envolvente más cercano cuya propiedad `position` no sea `static`, o con relación al documento si dicho elemento envolvente no existe.

Podemos utilizar esto para crear una animación. El siguiente documento despliega una imagen de un gato que se mueve alrededor de una elipse:

```
<p style="text-align: center">
  <img src="img/cat.png" style="position: relative">
</p>
<script>
  let gato = document.querySelector("img");
  let angulo = Math.PI / 2;
  function animar(tiempo, ultimoTiempo) {
    if (ultimoTiempo != null) {
      angulo += (tiempo - ultimoTiempo) * 0.001;
    }
    gato.style.top = (Math.sin(angulo) * 20) + "px";
    gato.style.left = (Math.cos(angulo) * 200) + "px";
    requestAnimationFrame(nuevoTiempo => animar(nuevoTiempo, tiempo)
       );
  }
  requestAnimationFrame(animar);
</script>
```

La flecha gris indica el camino por el que se mueve la imagen.



Nuestra imagen se centra en la página y se le da un valor para `position` de `relative`. Actualizaremos repetidamente los estilos `top` y `left` de la imagen para moverla.

El script utiliza `requestAnimationFrame` para programar la función `animar` para ejecutarse en el momento en el que el navegador está listo para volver a pintar la pantalla. La misma función `animar` llama a `requestAnimationFrame` otra vez para programar la siguiente actualización. Cuando la ventana del navegador (o pestaña) está activa, esto causará que sucedan actualizaciones a un rango de aproximadamente 60 actualizaciones por segundo, lo que tiende a producir una animación agradable a la vista.

Si únicamente actualizáramos el DOM en un ciclo, la página se congelaría, y no se mostraría nada en la pantalla. Los navegadores no actualizan la pantalla si un programa de JavaScript se encuentra en ejecución, tampoco permiten ninguna interacción con la página. Es por esto que necesitamos a

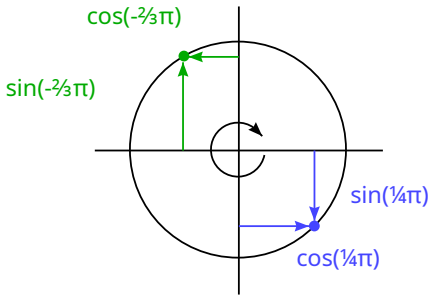
`requestAnimationFrame`—le permite al navegador saber que hemos terminado por el momento, y que puede empezar a hacer las cosas que le navegador hace, cómo actualizar la pantalla y responder a las acciones del usuario.

A la función `animar` se le pasa el tiempo actual como un argumento. Para asegurarse de que el movimiento del gato por milisegundo es estable, basa la velocidad a la que cambia el ángulo en la diferencia entre el tiempo actual y la última vez que la función se ejecutó. Si solamente movieramos el ángulo una cierta cantidad por paso, la animación tartamudearía si, por ejemplo, otra tarea pesada se encontrara ejecutándose en la misma computadora que pudiera prevenir que la función se ejecutará por una fracción de segundo.

Moverse en círculos se logra a través de las funciones `Math.cos` y `Math.sin`. Para aquellos que no estén familiarizados con estas, las introduciré brevemente dado que las usaremos ocasionalmente en este libro.

Las funciones `Math.cos` y `Math.sin` son útiles para encontrar puntos que recaen en un círculo alrededor del punto (0,0) con un radio de uno. Ambas funciones interpretan sus argumentos como las posiciones en el círculo, con cero denotando el punto en la parte más alejada del lado derecho del círculo, moviéndose en el sentido de las manecillas del reloj hasta que 2π (cerca de 6.28) nos halla tomado alrededor de todo el círculo. `Math.cos` indica la coordenada x del punto que corresponde con la posición dada, y `Math.sin` indica la coordenada y. Las posiciones (o ángulos) mayores que 2π o menores que 0 son válidas—la rotación se repite por lo que $a+2\pi$ se refiere al mismo ángulo que a.

Esta unidad para medir ángulos se conoce como radianes-un círculo completo corresponde a 2π radianes, de manera similar a 360 grados cuando se utilizan grados. La constante π está disponible como `Math.PI` en JavaScript.



El código de animación del gato mantiene un contador, `angulo`, para el ángulo actual de la animación y lo incrementa cada vez que la función `animar` es llamada. Luego, se puede utilizar este ángulo para calcular la posición actual del elemento imagen. El estilo `top` es calculado con `Math.sin` y multiplicado por 20, que es el radio vertical de nuestra elipse. El estilo `left` se basa en

`Math.cos` multiplicado por 200 por lo que la elipse es mucho más ancha que su altura.

Nótese que los estilos usualmente necesitan *unidades*. En este caso, tuvimos que agregar `"px"` al número para informarle al navegador que estábamos contando en pixeles (al contrario de centímetros, "ems", u otras unidades). Esto es sencillo de olvidar. Usar números sin unidades resultará en estilos que son ignorados—a menos que el número sea 0, que siempre indica la misma cosa, independientemente de su unidad.

Resumen

Los programas de JavaScript pueden inspeccionar e interferir con el documento que el navegador está desplegando a través de una estructura de datos llamada el DOM. Esta estructura de datos representa el modelo del navegador del documento, y un programa de JavaScript puede modificarlo para cambiar el documento visible.

El DOM está organizado como un árbol, en el cual los elementos están ordenados jerárquicamente de acuerdo a la estructura del documento. Estos objetos que representan a los elementos tienen propiedades como `parentNode` y `childNodes`, que pueden ser usadas para navegar a través de este árbol.

La forma en que un documento es desplegado puede ser influenciada por la *estilización*, tanto agregando estilos directamente a los nodos cómo definiendo reglas que coincidan con ciertos nodos. Hay muchas propiedades de estilo diferentes, tales como `color` o `display`. El código de JavaScript puede manipular el estilo de un elemento directamente a través de su propiedad de `style`.

Ejercicios

Construye una tabla

Una tabla HTML se construye con la siguiente estructura de etiquetas:

```
<table>
  <tr>
    <th>nombre</th>
    <th>altura</th>
    <th>ubicacion</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
```

```
    <td>Tanzania</td>
  </tr>
</table>
```

Para cada *fila*, la etiqueta `<table>` contiene una etiqueta `<tr>`. Dentro de estas etiquetas `<tr>`, podemos poner ciertos elementos: ya sean celdas cabecera (`<th>`) o celdas regulares (`<td>`).

Dado un conjunto de datos de montañas, un arreglo de objetos con propiedades `nombre`, `altura` y `lugar`, genera la estructura DOM para una tabla que enlista esos objetos. Deberá tener una columna por llave y una fila por objeto, además de una fila cabecera con elementos `<th>` en la parte superior, listando los nombres de las columnas.

Escribe esto de manera que las columnas se deriven automáticamente de los objetos, tomando los nombres de propiedad del primer objeto en los datos.

Agrega la tabla resultante al elemento con el atributo `id` de `"montañas"` de manera que se vuelva visible en el documento.

Una vez que lo tengas funcionando, alinea a la derecha las celdas que contienen valores numéricos, estableciendo su propiedad `style.textAlign` cómo `"right"`.

Elementos por nombre de tag

El método `document.getElementsByTagName` regresa todos los elementos hijo con un nombre de etiqueta dado. Implementa tu propia versión de esto como una función que toma un nodo y una cadena (el nombre de la etiqueta) como argumentos y regresa un arreglo que contiene todos los nodos elemento descendientes con el nombre del tag dado.

Para encontrar el nombre del tag de un elemento, utiliza su propiedad `nodeName`. Pero considera que esto regresará el nombre de la etiqueta todo en mayúsculas. Utiliza las funciones de las cadenas (`string`), `toLowerCase` o `toUpperCase` para compensar esta situación.

El sombrero del gato

Extiende la animación del gato definida anteriormente de manera de que tanto el gato como su sombrero (`<img src="img/hat.png">`) orbiten en lados opuestos de la elipse.

O haz que el sombrero circule alrededor del gato. O altera la animación en alguna otra forma interesante.

Para hacer el posicionamiento de múltiples objetos más sencillo, probablemente sea buena idea intercambiar a un posicionamiento absoluto. Esto significa que `top` y `left` serán contados con relación a la parte superior izquierda del documento. Para evitar usar coordenadas negativas, que pueden causar que la imagen se mueva fuera de la página visible, puedes agregar un número fijo de pixeles a los valores de las posiciones.

"Tienes poder sobre tu mente, no sobre los acontecimientos. Date cuenta de esto, y encontrarás la fuerza."

—Marco Aurelio, Meditaciones

Chapter 14

Manejo de Eventos

Algunos programas funcionan con la entrada directa del usuario, como las acciones del mouse y el teclado. Ese tipo de entrada no está disponible como una estructura de datos bien organizada, viene pieza por pieza, en tiempo real, y se espera que el programa responda a ella a medida que sucede.

Manejador de eventos

Imagina una interfaz en donde la única forma de saber si una tecla del teclado está siendo presionada es leer el estado actual de esa tecla. Para poder reaccionar a las pulsaciones de teclas, tendrías que leer constantemente el estado de la tecla para poder detectarla antes de que se vuelva a soltar. Esto sería peligroso al realizar otros cálculos que requieran mucho tiempo, ya que se podría perder una pulsación de tecla.

Algunas máquinas antiguas manejan las entradas de esa forma. Un paso adelante de esto sería que el hardware o el sistema operativo detectaran la pulsación de la tecla y lo pusieran en una cola. Luego, un programa puede verificar periódicamente la cola por nuevos eventos y reaccionar a lo que encuentre allí.

Por supuesto, este tiene que recordar de mirar la cola, y hacerlo con frecuencia, porque en cualquier momento entre que se presione la tecla y que el programa se de cuenta del evento causará que que el programa no responda. Este enfoque es llamado *sondeo*. La mayororía de los programadores prefieren evitarlo.

Un mejor mecanismo es que el sistema notifique activamente a nuestro código cuando un evento ocurre. Los navegadores hacen esto permitiéndonos registrar funciones como *manejadores* (*manejadores*) para eventos específicos.

```
<p>Da clic en este documento para activar el manejador.</p>
<script>
  window.addEventListener("click", () => {
    console.log("¿Tocaste?");
  });
```

```
</script>
```

La *vinculación* `window` se refiere a un objeto integrado proporcionado por el navegador. Este representa la ventana del navegador que contiene el documento. Llamando a su método `addEventListener` se registra el segundo argumento que se llamará siempre que ocurra el evento descrito por su primer argumento.

Eventos y nodos DOM

Cada manejador de eventos del navegador es registrado dentro de un contexto. En el ejemplo anterior llamamos a `addEventListener` en el objeto `window` para registrar un manejador para toda la ventana. Este método puede también ser encontrado en elementos DOM y en algunos otros tipos de objetos. Los controladores de eventos son llamados únicamente cuando el evento ocurra en el contexto del objeto en que están registrados.

```
<button>Presióname</button>
<p>No hay manejadores aquí.</p>
<script>
  let boton = document.querySelector("button");
  boton.addEventListener("click", () => {
    console.log("Botón presionado.");
  });
</script>
```

Este ejemplo adjunta un manejador al nodo del botón. Los clics sobre el botón hacen que se ejecute ese manejador, pero los clics sobre el resto del documento no.

Dar a un nodo un atributo `onclick` tiene un efecto similar. Esto funciona para la mayoría de tipos de eventos—se puede adjuntar un manejador a través del atributo cuyo nombre es el nombre del evento con `on` en frente de este.

Pero un nodo puede tener únicamente un atributo `onclick`, por lo que se puede registrar únicamente un manejador por nodo de esa manera. El método `addEventListener` permite agregar cualquier número de manejadores siendo seguro agregar manejadores incluso si ya hay otro manejador en el elemento.

El método `removeEventListener`, llamado con argumentos similares a `addEventListener`, remueve un manejador:

```
<button>Botón de única acción</button>
```

```
<script>
  let boton = document.querySelector("button");
  function unaVez() {
    console.log("Hecho.");
    boton.removeEventListener("click", unaVez);
  }
  boton.addEventListener("click", unaVez);
</script>
```

La función dada a `removeEventListener` tiene que ser el mismo valor de función que se le dio a `addEventListener`. Entonces, para desregistrar un manejador, se le tiene que dar un nombre a la función (`unaVez`, en el ejemplo) para poder pasar el mismo valor de función a ambos métodos.

Objetos de evento

Aunque lo hemos ignorado hasta ahora, las funciones del manejador de eventos reciben un argumento: el *objeto de evento*. Este objeto contiene información adicional acerca del evento. Por ejemplo, si queremos saber *cuál* botón del mouse fue presionado, se puede ver la propiedad `button` del objeto de evento.

```
<button>Haz clic en mí de la forma que quieras</button>
<script>
  let boton = document.querySelector("button");
  boton.addEventListener("mousedown", evento => {
    if (evento.button == 0) {
      console.log("Botón izquierdo");
    } else if (evento.button == 1) {
      console.log("Botón central");
    } else if (evento.button == 2) {
      console.log("Botón derecho");
    }
  });
</script>
```

La información almacenada en un objeto de evento es diferente por cada tipo de evento. Se discutirán los distintos tipos de eventos más adelante en el capítulo. La propiedad `type` del objeto siempre contiene una cadena que identifica al evento (como `"click"` o `"mousedown"`)

Propagación

Para la mayoría de tipos de eventos, los manejadores registrados en nodos con hijos también recibirán los eventos que sucedan en los hijos. Si se hace clic a un botón dentro de un párrafo, los manejadores de eventos del párrafo también verán el evento clic.

Pero si tanto el párrafo como el botón tienen un manejador, el manejador más específico—el del botón—es el primero en lanzarse. Se dice que el evento se *propaga* hacia afuera, desde el nodo donde este sucedió hasta el nodo padre del nodo y hasta la raíz del documento. Finalmente, después de que todos los manejadores registrados en un nodo específico hayan tenido su turno, los manejadores registrados en general ventana tienen la oportunidad de responder al evento.

En cualquier momento, un manejador de eventos puede llamar al método `stopPropagation` en el objeto de evento para evitar que los manejadores que se encuentran más arriba reciban el evento. Esto puede ser útil cuando, por ejemplo, si tienes un botón dentro de otro elemento en el que se puede hacer clic y que no se quiere que los clics sobre el botón activen el comportamiento de clic del elemento exterior.

El siguiente ejemplo registra manejadores `"mousedown"` tanto en un botón como el párrafo que lo rodea. Cuando se hace clic con el botón derecho del mouse, el manejador del botón llama a `stopPropagation`, lo que evitará que se ejecute el manejador del párrafo. Cuando se hace clic en el botón con otro botón del mouse, ambos manejadores se ejecutarán.

```
<p>Un párrafo con un <button>botón</button>.</p>
<script>
  let parrafo = document.querySelector("p");
  let boton = document.querySelector("button");
  parrafo.addEventListener("mousedown", () => {
    console.log("Manejador del párrafo.");
  });
  boton.addEventListener("mousedown", evento => {
    console.log("Manejador del botón.");
    if (evento.button == 2) evento.stopPropagation();
  });
</script>
```

La mayoría de objetos de eventos tienen una propiedad `target` que se refiere al nodo donde se originaron. Se puede usar esta propiedad para asegurar de que no se está manejando accidentalmente algo que se propagó desde un nodo

que no se desea manejar.

También es posible utilizar la propiedad `target` para lanzar una red amplia para un evento específico. Por ejemplo, si tienes un nodo que contiene una gran cantidad de botones, puede ser más conveniente el registrar un manejador en un solo clic en el nodo externo y hacer que use la propiedad `target` para averiguar si se hizo clic en un botón, en lugar de registrar manejadores individuales en todos los botones.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", evento => {
    if (evento.target.nodeName == "BUTTON") {
      console.log("Presionado", evento.target.textContent);
    }
  });
</script>
```

Acciones por defecto

La mayoría de eventos tienen una acción por defecto asociada a ellos. Si haces clic en un enlace, se te dirigirá al destino del enlace. Si presionas la flecha hacia abajo, el navegador desplazará la página hacia abajo. Si das clic derecho, se obtendrá un menú contextual. Y así.

Para la mayoría de los tipos de eventos, los manejadores de eventos de JavaScript se llamarán *antes* de que el comportamiento por defecto se produzca. Si el manejador no quiere que suceda este comportamiento por defecto, normalmente porque ya se ha encargado de manejar el evento, se puede llamar al método `preventDefault` en el objeto de evento.

Esto puede ser utilizado para implementar un atajo de teclado propio o menú contextual. Esto también puede ser utilizado para interferir de forma desagradable el comportamiento que los usuarios esperan. Por ejemplo, aquí hay un enlace que no se puede seguir:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let enlace = document.querySelector("a");
  enlace.addEventListener("click", evento => {
    console.log("Nope.");
    evento.preventDefault();
```

```
  });
</script>
```

Trata de no hacer tales cosas a menos que tengas una buena razón para hacerlo. Será desagradable para las personas que usan tu página cuando el comportamiento esperado no funcione.

Dependiendo del navegador, algunos eventos no pueden ser interceptados en lo absoluto. En Chrome, por ejemplo, el atajo de teclado para cerrar la pestaña actual (CONTROL-W o COMMAND-W) no se puede manejar con JavaScript.

Eventos de teclado

Cuando una tecla del teclado es presionado, el navegador lanza un evento "`keydown`". Cuando este es liberado, se obtiene un evento `"keyup"`.

```
<p>Esta página se pone violenta cuando se mantiene presionado la
   tecla V.</p>
<script>
  window.addEventListener("keydown", evento => {
    if (evento.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", evento => {
    if (evento.key == "v") {
      document.body.style.background = "";
    }
  });
</script>
```

A pesar de su nombre, `"keydown"` se lanza no solamente cuando la tecla es físicamente presionada. Cuando una tecla se presiona y se mantiene presionada, el evento se lanza una vez más cada que la tecla se *repite*. Algunas veces se debe tener cuidado con esto. Por ejemplo, si tienes un botón al DOM cuando el botón es presionado y removido cuando la tecla es liberada, puedes agregar accidentalmente cientos de botones cuando la tecla se mantiene presionada por más tiempo.

El ejemplo analizó la propiedad `key` del objeto de evento para ver de qué tecla se trata el evento. Esta propiedad contiene una cadena que, para la mayoría de las teclas, corresponde a lo que escribiría al presionar esa tecla. Para teclas especiales como ENTER, este contiene una cadena que nombre la tecla {`"Enter`

", en este caso}. Si mantienes presionado SHIFT mientras que presionas una tecla, esto también puede influir en el nombre de la tecla-`"v"` se convierte en `"V"` y `"1"` puede convertirse en `"!"`, es lo que se produce al presionar SHIFT-1 en tu teclado.

La teclas modificadoras como SHIFT, CONTROL, ALT y META (COMMAND en Mac) generan eventos de teclado justamente como las teclas normales. Pero cuando se busque combinaciones de teclas, también se puede averiguar si estas teclas se mantienen presionadas viendo las propiedades `shiftKey`, `ctrlKey`, `altKey` y `metaKey` de los eventos de teclado y mouse.

```
<p>Presiona Control-Espacio para continuar.</p>
<script>
  window.addEventListener("keydown", evento => {
    if (evento.key == " " && evento.ctrlKey) {
      console.log("¡Continuando!");
    }
  });
</script>
```

El nodo DOM donde un evento de teclado se origina depende en el elemento que tiene el foco cuando la tecla es presionada. La mayoría de los nodos no pueden tener el foco a menos que se les de un atributo `tabindex`, pero elementos como enlaces, botones y campos de formularios sí pueden. Volveremos a los campos de formularios en el capítulo Chapter?. Cuando nadie en particular tiene el foco, `document.body` actua como el nodo objetivo de los eventos de teclado.

Cuando el usuario está escribiendo texto, usando los eventos de teclado para averiguar qué se está escribiendo es problematico. Algunas plataformas, sobre todo el teclado virtual en teléfonos Android, no lanzan eventos de teclado. Pero incluso cuando se tiene un teclado antiguo, algunos tipos de entradas de texto no coinciden con las pulsaciones de teclas de forma sencilla, como el software *editor de métodos de entrada* (IME) usado por personas cuyos *guiones* (*script*) no encajan en un teclado, donde se combinan varias teclas para crear caracteres.

Para notar cuando se escribió algo, los elementos en los que se puede escribir, como las etiquetas `<input>` y `<textarea>`, lanzan eventos `"input"` cada vez que el usuario cambia su contenido. Para obtener el contenido real que fue escrito, es mejor leerlo directamente desde el campo seleccionado. El Chapter ? mostrará cómo.

Eventos de puntero

Actualmente, hay dos formas muy utilizadas para señalar en una pantalla: mouse (incluyendo dispositivos que actuan como mouse, como páneles táctiles y *trackballs*) y pantallas táctiles. Estos producen diferentes tipos de eventos.

Clics de mouse

Al presionar un botón del mouse se lanzan varios eventos. Los eventos "`mousedown`" y "`mouseup`" son similares a "`keydown`" y "`keyup`" y se lanzan cuando el boton es presionado y liberado. Estos ocurren en los nodos del DOM que están inmediatamente debajo del puntero del mouse cuando ocurre el evento.

Después del evento "`mouseup`", un evento "`click`" es lanzado en el nodo más específico que contiene la pulsación y la liberación del botón. Por ejemplo, si presiono el botón del mouse sobre un párrafo y entonces muevo el puntero a otro párrafo y suelto el botón, el evento "`click`" ocurrirá en el elemento que contiene ambos párrafos.

Si se producen dos clics juntos, un evento "`dblclick`" (doble-clic) también se lanza, después del segundo evento de clic.

Para obtener la información precisa sobre el lugar en donde un evento del mouse ocurrió, se puede ver en las propiedades `clientX` y `clientY`, los cuales contienen las coordenadas (en pixeles) relativas a la esquina superior izquierda de la ventana o `pageX` y `pageY`, las cuales son relativas a la esquina superior izquierda de todo el documento (el cual puede ser diferente cuando la ventana ha sido desplazada).

Lo siguiente implementa un programa de dibujo primitivo. Cada vez que se hace clic en el documento, agrega un punto debajo del puntero del mouse. Ver Chapter ? para un programa de dibujo menos primitivo.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .punto {
    height: 8px; width: 8px;
    border-radius: 4px; /* redondea las esquinas */
    background: blue;
    position: absolute;
  }
</style>
<script>
```

```
  window.addEventListener("click", evento => {
    let punto = document.createElement("div");
    punto.className = "punto";
    punto.style.left = (evento.pageX - 4) + "px";
    punto.style.top = (evento.pageY - 4) + "px";
    document.body.appendChild(punto);
  });
</script>
```

Movimiento del mouse

Cada vez que el puntero del mouse se mueve, un evento `"mousemove"` es lanzado. Este evento puede ser utilizado para rastrear la posición del mouse. Una situación común en la cual es útil es cuando se implementa una funcionalidad de mouse-arrastrar.

Como un ejemplo, el siguiente programa muestra una barra y configura los manejadores de eventos para que cuando se arrastre hacia la izquierda o a la derecha en esta barra la haga más estrecha o más ancha:

```
<p>Dibuja la barra para cambiar su anchura:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  let ultimoX; // Rastrea la última posición de X del mouse
     observado
  let barra = document.querySelector("div");
  barra.addEventListener("mousedown", evento => {
    if (evento.button == 0) {
      ultimoX = evento.clientX;
      window.addEventListener("mousemove", movido);
      evento.preventDefault(); // Evitar la selección
    }
  });

  function movido(evento) {
    if (evento.buttons == 0) {
      window.removeEventListener("mousemove", movido);
    } else {
      let distancia = event.clientX - lastX;
      let nuevaAnchura = Math.max(10, barra.offsetWidth + distancia)
         ;
      barra.style.width = nuevaAnchura + "px";
      ultimoX = evento.clientX;
```

```
    }
  }
</script>
```

La página resultante se ve así:

{{figure {url: "img/drag-bar.png", alt: "Una barra arrastrable", width: "5.3cm"}}}

Tener en cuenta que el manejador `"mousemove"` es registrado sobre toda la ventana. Incluso si el mouse se sale de la barra durante el cambio de tamaño, siempre que se mantenga presionado el botón, su tamaño se actualizará.

Se debe dejar de cambiar el tamaño de la barra cuando se suelta el botón del mouse. Para eso, podemos usar la propiedad `buttons` (nótese el plural), que informa sobre los botones que se mantienen presionados actualmente. Cuando este es cero, no hay botones presionados. Cuando los botones se mantienen presionados, su valor es la suma de los códigos de esos botones—el botón izquierdo tiene el código 1, el botón derecho 2 y el central 4. De esta manera, puedes verificar si un botón dado es presionado tomando el resto del valor de `buttons` y su código.

Tener en cuenta que el orden de los códigos es diferente del utilizado por `button`, donde el botón central se encuentra que el derecho. Como se mencionó, la coherencia no es realmente un punto fuerte de la interfaz de programación del navegador.

Eventos de toques

El estilo del navegador gráfico que usamos fue diseñado con la interfaz de mouse en mente, en un tiempo en el cual las pantallas táctiles eran raras. Para hacer que la Web "funcionara" en los primeros teléfonos con pantalla táctil, los navegadores de esos dispositivos pretendían, hasta cierto punto, que los eventos táctiles fueran eventos del mouse. Si se toca la pantalla, se obtendrán los eventos `"mousedown"`, `"mouseup"` y `"click"`.

Pero esta ilusión no es muy robusta. Una pantalla táctil funciona de manera diferente a un mouse: no tiene multiples botones, no puedes rastrear el dedo cuando no está en la pantalla (para simular `"mousemove"`) y permite que multiples dedos estén en la pantalla al mismo tiempo.

Los eventos del mouse cubren las interacciones solamente en casos sencillos—si se agrega un manejador `"click"` a un butón, los usuarios táctiles aún podrán usarlo. Pero algo como la barra redimensionable del ejemplo anterior no funciona en una pantalla táctil.

Hay tipos de eventos específicos lanzados por la interacción táctil. Cuando un dedo inicia tocando la pantalla, se obtiene el evento `"touchstart"`. Cuando este es movido mientras se toca, se lanza el evento `"touchmove"`. Finalmente, cuando se deja de tocar la pantalla, se lanzará un evento `"touchend"`.

Debido a que muchas pantallas táctiles pueden detectar multiples dedos al mismo tiempo, estos eventos no tienen un solo conjunto de coordenadas asociados a ellos. Más bien, sus objetos de evento tienen una propiedad `touches`, el cual contiene un objeto tipo matriz de puntos, cada uno de ellos tiene sus propias propiedades `clientX`, `clientY`, `pageX` y `pageY`.

Se puede hacer algo como esto para mostrar circulos rojos alrededor de cada que toca:

```
<style>
  punto { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Toca esta página</p>
<script>
  function actualizar(event) {
    for (let punto; punto = document.querySelector("punto");) {
      punto.remove();
    }
    for (let i = 0; i < evento.touches.length; i++) {
      let {paginaX, paginaY} = evento.touches[i];
      let punto = document.createElement("punto");
      punto.style.left = (paginaX - 50) + "px";
      punto.style.top = (paginaY - 50) + "px";
      document.body.appendChild(punto);
    }
  }
  window.addEventListener("touchstart", actualizar);
  window.addEventListener("touchmove", actualizar);
  window.addEventListener("touchend", actualizar);
</script>
```

A menudo se querrá llamar a `preventDefault` en un manejador de eventos táctiles para sobreescribir el comportamiento por defecto del navegador (que puede incluir desplazarse por la pagína al deslizar el dedo) y para evitar que los eventos del mouse se lancen, para lo cual se puede tener *también* un manejador.

Eventos de desplazamiento

Siempre que un elemento es desplazado, un evento `"scroll"` es lanzado. Esto tiene varios usos, como son saber qué está mirando el usuario actualmente (para deshabilitar la animación fuera de pantalla o enviar informes espías a su sede maligna) o mostrar alguna indicación de progreso (resaltando parte de una tabla de contenido o mostrando un número de página).

El siguiente ejemplo dibuja una barra de progreso sobre el documento y lo actualiza para que se llene a medida que se desplza hacia abajo:

```
<style>
  #progreso {
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progreso"></div>
<script>
  // Crear algo de contenido
  document.body.appendChild(document.createTextNode(
    "supercalifragilisticoespialidoso ".repeat(1000)));

  let barra = document.querySelector("#progreso");
  window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight;
    barra.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>
```

Darle a un elemento una `position` de `fixed` actua como una posición `absolute` pero también evita que se desplace junto con el resto del documento. El efecto es hacer que la barra de progreso permanezca en la parte superior. Su ancho cambia para indicar el progreso actual. Se usa `%`, en lugar de `px`, como una unidad cuando se configura el ancho para que el tamaño del elemento sea relativo al ancho de la página.

La vincualación global `innerHeight` proporciona la altura de la ventana, que se tiene que restar de la altura total desplazable—no se puede seguir desplazando cuando se llega al final del documento. También hay un `innerWidth` para el ancho de la ventana. Al dividir `pageYOffset`, la posición de desplazamiento actual, por la posición de desplazamiento máximo y mulplicado por

100, se obtiene el porcentaje de la barra de progreso.

Al llamar `preventDefault` en un evento de desplazamiento no evita que el desplazamiento se produzca. De hecho, el manejador de eventos es llamado unicamente *después* de que se realiza el desplazamiento.

Eventos de foco

Cuando un elemento gana el foco, el navegador lanza un evento de `"focus"` en él. Cuando este pierde el foco, el elemento obtiene un evento `"blur"`.

A diferencia de los eventos discutidos con anterioridad, estos dos eventos no se propagan. Un manejador en un elemento padre no es notificado cuando un elemento hijo gana o pierde el foco.

El siguiente ejemplo muestra un texto de ayuda para campo de texto que actualmente tiene el foco:

```
<p>Nombre: <input type="text" dato-ayuda="Tu nombre"></p>
<p>Edad: <input type="text" dato-ayuda="Tu edad en años"></p>
<p id="ayuda"></p>

<script>
  let ayuda = document.querySelector("#ayuda");
  let campos = document.querySelectorAll("input");
  for (let campo of Array.from(campos)) {
    campo.addEventListener("focus", evento => {
      let texto = evento.target.getAttribute("dato-ayuda");
      ayuda.textContent = texto;
    });
    field.addEventListener("blur", evento => {
      ayuda.textContent = "";
    });
  }
</script>
```

Esta captura de pantalla muestra el texto de ayuda para el campo edad.

{{figure {url: “img/help-field.png”, alt: “Brindar ayuda cuando un campo está enfocado”, width: “4.4cm”}}}

El objeto window recibirá eventos de `"focus"` y `"blur"` cuando el usuario se mueva desde o hacia la pestaña o ventana del navegador en la que se muestra el documento.

Evento de carga

Cuando una página termina de cargarse, el evento `"load"` es lanzado en la ventana y en los objetos del cuerpo del documento. Esto es usado a menudo para programar acciones de inicialización que requieren que todo el documento haya sido construido. Recordar que el contenido de las etiquetas `<script>` se ejecutan inmediatamente cuando la etiqueta es encontrada. Esto puede ser demasiado pronto, por ejemplo cuando el guión necesita hacer algo con las partes del documento que aparecen después de la etiqueta `<script>`.

Elementos como imagenes y etiquetas de guiones que cargan un archivo externo también tienen un evento `"load"` que indica que se cargaron los archivos a los que hacen referencia. Al igual que los eventos relacionado con el foco, los eventos de carga no se propagan.

Cuando una página se cierra o se navega fuera de ella (por ejemplo, siguiendo un enlace), un evento `"beforeunload"` es lanzado. El principal uso de este evento es evitar que el usuario pierda su trabajo accidentalmente al cerrar un documento. Si se evita el comportamiento por defecto en este evento *y* se establece la propiedad `returnValue` en el objeto de evento a una cadena, el navegador le mostrará al usuario un diálogo preguntándole si realmente quiere salir de la página. Ese cuadro de diálogo puede incluir una cadena de texto, pero debido a que algunos sitios intentan usar estos cuadros de diálogo para confundir a las personas para que permanezcan en su página para ver anuncios poco fiables sobre la pérdida de peso, la mayoría de los navegadores ya no lo muestran.

Eventos y el ciclo de eventos

En el contexto del ciclo de eventos, como se explica en el Chapter 11, los manejadores de eventos del navegador se comportan como otras notificaciones asíncronas. Se programan cuando el evento ocurre pero deben esperar a que finalicen otros guiones que se están ejecutando antes de que tengan la oportunidad de ejecutarse.

El hecho de que los eventos puedan ser procesados solo cuando no se está ejecutando nada más que, si el ciclo de eventos está vinculado con otro trabajo, cualquier interacción con la página (que ocurre a través de eventos) se retrasará hasta que haya tiempo para procesarlo. Por lo tanto, si programas demasiado trabajo, ya sea con manejadores de eventos de ejecución prolongada o con muchos de breve ejecución, la página se volverá lenta y engorrosa de usar.

Para casos en donde *realmente* se quiera hacer algo que requiera mucho

tiempo en segundo plano sin congelar la página, los navegadores proveeen algo llamado **web worker**. Un worker es un proceso de JavaScript que se jecuta junto con el guión principal, en su propia línea de tiempo.

Imaginar que se eleva un número al cuadrado es un cálculo pesado y de larga duración que se quiere realizar en un hilo separado. Se podría escribir un archivo llamado `code/cuadradoworker.js` que responde a los mensajes calculando un cuadrado y enviando de vuelta un mensaje.

```
addEventListener("message", evento => {
  enviarMensaje(evento.data * evento.data);
});
```

Para evitar el problema de tener multiples hilos tocando los mismos datos, los workers no comparten su alcance global o cualquier otro tipo de dato con el entorno del guión principal. En cambio, tienes que comunicarte con ellos enviando mensajes de un lado a otro.

Este código genera un worker que ejecuta ese guión, le envía algunos mensajes y genera las respuestas.

```
let cuadradoWorker = new Worker("code/cuadradoworker.js");
cuadradoWorker.addEventListener("message", evento => {
  console.log("El respondió:", evento.data);
});
cuadradoWorker.enviarMensaje(10);
cuadradoWorker.enviarMensaje(24);
```

La función `enviarMensaje` envía un mensaje, que provocará que se lance un evento `"message"`en el receptor. El guión que creó al worker envía y recibe mensajes a través del objeto `Worker`, mientras que el worker habla con el guión que lo creó enviando y escuchando directamente en su alcance global. Solo los valores que pueden ser representados como un JSON pueden ser enviados como un mensaje—el otro lado recibirá una *copia* de ellos, en lugar del valor en sí.

Temporizadores

Se mostró la función `establecerTiempoEspera` en el Chapter 11. Este programa otra función para que se llame más tarde, después de un número determinado de milisegundos.

Algunas veces se necesita cancelar una función que se haya programado. Esto se hace almacenando el valor retornado por `establecerTiempoEspera` y

llamando a `reinicarTiempoEspera` en él.

```
let temporizadorBomba = setTimeout(() => {
  console.log("¡BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% de cambio
  console.log("Desactivada.");
  reinicarTiempoEspera(temporizadorBomba);
}
```

La función `cancelAnimationFrame` funciona de la misma forma que `reinicarTiempoEspera` —llamarla en un valor devuelto por `requestAnimationFrame` cancelará ese marco (asumiendo que aún no ha sido llamado).

Un conjunto similar de funciones, `setInterval` y `clearInterval`, son usadas para restablecer los temporizadores que deberían *repetirse* cada X milisegundos.

```
let tictac = 0;
let reloj = setInterval(() => {
  console.log("tictac", tictac++);
  if (tictac == 10) {
    clearInterval(reloj);
    console.log("Detener.");
  }
}, 200);
```

Antirrebote

Algunos tipos de eventos tienen el potencial de ser lanzados rápidamente, muchas veces seguidos (los eventos `"mousemove"` y `"scroll"`, por ejmplo). Cuando se menejan tales eventos, se debe tener cuidado de no hacer nada que consuma demasiado tiempo o su manejador tomará tanto tiempo que la interacción con el documento comenzará a sentirse lenta.

Si necesitas hacer algo no trivial en algún manejador de este tipo, se puede usar `setTimeout` para asegurarse de que no se está haciendo con demasiada frecuencia. Esto generalmente se llama *antirrebote* del evento. Hay varios enfoques ligeramente diferentes para esto.

En el primer ejemplo, se quiere reaccionar cuando el usuario ha escrito algo, pero no se quiere hacer inmediatamente por cada evento de entrada. Cuando se está escribiendo rápidamente, se requiere esperar hasta que se produzca una

pausa. En lugar de realizar inmediatamente una acción en el manejador de eventos, se establece un tiempo de espera. También se borra el tiempo de espera anterior (si lo hay) para que cuando los eventos ocurran muy juntos (más cerca que el tiempo de espera), el tiempo de espera del evento anterior será cancelado.

```
<textarea>Escribe algo aquí...</textarea>
<script>
  let areaTexto = document.querySelector("textarea");
  let tiempoEspera;
  areaTexto.addEventListener("input", () => {
    clearTimeout(tiempoEspera);
    tiempoEspera = setTimeout(() => console.log("¡Escribió!"), 500);
  });
</script>
```

Dar un valor indefinido a `clearTimeout` o llamándolo en un tiempo de espera que ya ha se ha lanzado no tiene ningún efecto. Por lo tanto, no debemos tener cuidado sobre cuándo llamarlo, y simplemente se hace para cada evento.

Podemos utilizar un patrón ligeramente diferente si se quiere espaciar las respuestas de modo que estén separadas por al menos una cierta longitud de tiempo pero se quiere lanzar *durante* una serie de eventos, no solo después. Por ejemplo, se podría querer responder a los eventos `"mousemove"` mostrando las coordenadas actuales del mourse pero solo cada 250 milisegundos.

```
<script>
  let programado = null;
  window.addEventListener("mousemove", evento => {
    if (!programado) {
      setTimeout(() => {
        document.body.textContent =
          `Mouse en ${programado.pageX}, ${programado.pageY}`;
        programado = null;
      }, 250);
    }
    programado = evento;
  });
</script>
```

Resumen

Los manejadores de eventos hacen posible detectar y reaccionar a eventos que suceden en nuestra página web. El método `addEventListener` es usado para registrar un manejador de eventos.

Cada evento tiene un tipo (`"keydown"`, `"focus"`, etc.) que lo identifica. La mayoría de eventos son llamados en un elemento DOM específico y luego se *propagan* a los ancentros de ese elemento, lo que permite que los manejadores asociados con esos elementos los manejen.

Cuando un manejador de evento es llamado, se le pasa un objeto evento con información adicional acerca del evento. Este objeto también tiene métodos que permiten detener una mayor propagación (`stopPropagation`) y evitar que el navegador maneje el evento por defecto (`preventDefault`).

Al presiosar una tecla se lanza los eventos `"keydown"` y `"keyup"`. Al presionar un botón del mouse se lanzan los eventos `"mousedown"`, `"mouseup"` y `"click"` . Al mover el mouse se lanzan los eventos `"mousemove"`. Las interacción de la pantalla táctil darán como resultado eventos `"touchstart"`, `"touchmove"` y `"touchend"`.

El desplazamiento puede ser detectado con el evento `"scroll"` y los cambios de foco pueden ser detactados con los eventos `"focus"` y `"blur"`. Cuando el documento termina de cargarse, se lanza el evento `"load"` en la ventana.

Ejercicios

Globo

Escribe una página que muestre un globo (usando el globo emoji, 🎈). Cuando se presione la flecha hacia arriba, debe inflarse (crecer) un 10 por cierto, y cuando se presiona la flecha hacia abajo, debe desinflarse (contraerse) un 10 por cierto)

Puedes controlar el tamaño del texto (los emojis son texto) configurando la propiedad CSS `font-size` (`style.fontSize`) en su elemento padre. Recuerda incluir una unidad en el valor, por ejemplo pixeles (`10px`).

Los nombres de las teclas de flecha son `"ArrowUp"` y `"ArrowDown"`. Asegúratede que las teclas cambien solo al globo, sin desplazar la página.

Cuando eso funcione, agrega una nueva función en la que, si infla el globo más allá de cierto tamaño, explote. En este caso, explotar significa que se reemplaza con un emoji 💥, y se elimina el manejador de eventos (para que no se pueda inflar o desinflar la explosión).

Mouse trail

En los primeros días de JavaScript, que era el momento de páginas de inicio llamativas con muchas imágenes, a la gente se le ocurrieron formas realmente inspiradoras de usar el lenguaje.

Uno de estos fue el *rastro del mouse*, una serie de elementos que seguirían el puntero del mouse mientras lo movías por la página.

En este ejercicio, quiero que implementes un rastro del mouse. Utiliza elementos `<div>` con un tamaño fijo y un color de fondo (consulta a code en la sección "Clics del mouse" por un ejemplo). Crea un montón de estos elementos y, cuando el mouse se mueva, muestralos después del puntero del mouse.

Hay varios enfoques posibles aquí. Puedes hacer tu solución tan simple o tan compleja como desees. Una solución simple para comenzar es mantener un número de elementos de seguimiento fijos y recorrerlos, moviendo el sigueinte a la posición actual del mouse cada vez que ocurra un evento `"mousemove"`.

Pestañas

Los paneles con pestañas son utilizados ampliamente en las interfaces de usuario. Te permiten seleccionar un panel de interfaz eligiendo entre una serie de pestañas "que sobresalen" sobre un elemento.

En este ejercicio debes implementar una interfaz con pestañas simple. Escribe una función, `asTabs`, que tome un nodo DOM y cree una interfaz con pestañas que muestre los elementos secundarios de ese nodo. Se debe insertar una lista de elementos `<button>` en la parte superior del nodo, uno para cada elemento hijo, que contenga texto recuperado del atributo `data-tabname` del hijo. Todos menos uno de los elementos secundarios originales deben estar ocultos (dado un estilo `display` de`none`). El nodo visible actualmente se puede seleccionar haciendo clic en los botones.

Cuando eso funcione, extiéndelo para diseñar el botón de la pestaña seleccionada actualmente de manera diferente para que sea obvio qué pestaña está seleccionada.

"Toda realidad es un juego."

—Iain Banks, El Jugador de Juegos

Chapter 15

Proyecto: Un Juego de Plataforma

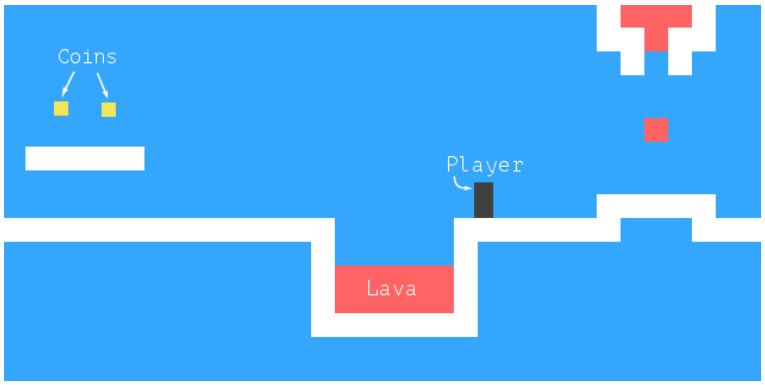
Mucha de mi fascinación inicial con computadoras, como la de muchos niños nerd, tiene que ver con juegos de computadora. Fui atraído a los pequeños mundos simulados que podía manipular y en los que las historias (algo así) revelaban más, yo supongo, por la manera en que yo proyectaba mi imaginación en ellas más que por las posibilidades que realmente ofrecían.

No le deseo a nadie una carrera en programación de juegos. Tal como la industria musical, la discrepancia entre el número de personas jóvenes ansiosas queriendo trabajar en ella y la actual demanda de tales personas crea un ambiente bastante malsano.

Este capítulo recorrerá la implementación de un juego de plataforma pequeño. Los juegos de plataforma (o juegos de "brincar y correr") son juegos que esperan que el jugador mueva una figura a través de un mundo, el cual usualmente es bidimensional y visto de lado, mientras brinca encima y sobre cosas.

El juego

Nuestro juego estará ligeramente basado en Dark Blue (*www.lessmilk.com/games/10*) de Thomas Palef. Escogí ese juego porque es tanto entretenido como minimalista y porque puede ser construido sin demasiado código. Se ve así:



La caja oscura representa al jugador, cuya tarea es recolectar las cajas amarillas (monedas) mientras evita la cosa roja (lava). Un nivel es completado cuando todas las monedas han sido recolectadas.

El jugador puede caminar alrededor con las teclas de las flechas izquierda y derecha y pueden brincar con la flecha arriba. Brincar es una especilidad de este personaje de juego. Puede alcanzar varias veces su propia altura y puede cambiar de dirección en el aire. Esto puede no ser completamente realista, pero ayuda a darle al jugador el sentimiento de estar en control directo del avatar.

El juego consiste en un fondo estático, expuesto como una cuadrícula, con los elementos móviles superpuestos sobre ese fondo. Cada campo en la cuadrícula está ya sea vacío, sólido o lava. Las posiciones de estos elementos no están restringidas a la cuadrícula-sus coordenadas pueden ser fraccionales, permitiendo un movimiento suave.

La tecnología

Usaremos el DOM del navegador para mostrar el juego, y leeremos la entrada del usuario por medio del manejo los eventos de teclas.

El código relacionado con la pantalla y el teclado es sólo una pequeña parte del trabajo que necesitamos hacer para construir este juego. Ya que todo parece como cajas coloreadas, dibujar es no es complicado: creamos los elementos del DOM y usamos estilos para darlos un color de fondo, tamaño y posición.

Podemos representar el fondo como una tabla ya que es una cuadrícula invariable de cuadrados. Los elementos libres de moverse pueden ser superpuestos usando elementos posicionados absolutamente.

En los juegos y otros programas que deben animar gráficos y responder a la entrada del usuario sin retraso notable, la eficiencia es importante. Aunque el DOM no fue diseñado originalmente para gráficos de alto rendimiento, es realmente mejor en ello de lo que se esperaría. Viste algunas animaciones en Chapter 13. En una máquina moderna, un simple juego como este desempeña bien, aún si no nos preocupamos mucho de la optimización.

En el siguiente capítulo, exploraremos otra tecnología del navegador, la etiqueta `<canvas>`, la cual provee un forma más tradicional de dibujar gráficos, trabajando en término de formas y pixeles más que elementos del DOM.

Niveles

Vamos a querer una forma de especificar niveles que sea fácilmente leíble y editable por humanos. Ya que está bien que todo empiece sobre una cuadrícula,

podríamos usar cadenas de caracteres largas en las que cada caracter representa un elemento-ya sea una parte de la cuadrícula de fondo o un elemento móvil.

El plano para un nivel pequeño podría lucir como esto:

```
let planoDeNivel = `
......................
..#................#..
..#..............=.#..
..#.........o.o....#..
..#.@......#####...#..
..#####............#..
......#++++++++++++#..
......##############..
......................`;
```

Los puntos son espacios vacíos, los signos de número (#) son muros y los signos de suma son lava. La posición inicial del jugador es el arroba (@). Cada caracter O es una moneda y el signo de igual (=) en la parte superior es un bloque de lava que se mueve de un lado a otro horizontalmente.

Vamos a soportar dos tipos adicionales de lava en movimento: el caracter de la barra vertical (|) crea gotas que mueven verticalmente y la v indica lava *goteando*-la lava que se mueve verticalmente que no rebota de un lado a otro sino que sólo se mueve hacia abajo, brincando a su posición inicial cuando toca el suelo.

Un juego completo consiste en múltiples niveles que el jugador debe completar. Un nivel es completado cuando todas las monedass han sido recolectadas. Si el jugador toca la lava, el nivel actual se restaura a su posición inicial y el juego puede intentar de nuevo.

Leyendo un nivel

La siguiente clase guarda un objeto de nivel. Su argumento debe ser la cadena de carateres que define el nivel.

```
class Nivel {
  constructor(plano) {
    let filas = plano.trim().split("\n").map(l => [...l]);
    this.height = filas.length;
    this.width = filas[0].length;
    this.iniciarActores = [];

    this.filas = filas.map((fila, y) => {
```

```
      return fila.map((car, x) => {
        let tipo = caracteresDelNivel[car];
        if (typeof tipo == "string") return tipo;
        this.iniciarActores.push(
          tipo.create(new Vector(x, y), car));
        return "vacío";
      });
    });
  }
}
```

El método `trim` es usado para remover el espacio en blanco al inicio y al final de la cadena de caracteres del plano. Esto permite a nuestro plano de ejemplo empezar con una nueva línea de manera que todas las líneas esten directamente una debajo de la otra. La cadena restante es dividida en caracteres de nueva línea y cada línea es propagada en un arreglo, produciendo un arreglo de caracteres.

Entonces `filas` guarda un arreglo de arreglos de caracteres, las filas del plano. Podemos derivar el ancho y alto del nivel de ellas. Pero aún debemos separar los elementos móviles de la cuadrícula de fondo. Llamaremos a los elementos móviles *actores*. Ellos se guardaran en un arreglo de objetos. El fondo será un arreglo de arreglos de caracteres, que tendrán tipos de campos como `"vacío"`, `"muro"` o `"lava"`.

Para crear estos arreglos, mapearemos sobre las filas y luego sobre su contenido. Recuerda que `map` pasa el índice del arreglo como un segundo argumento a la función de mapeo, que nos dice las coordenadas x y las coordenadas y de un caracter dado. Las posiciones en el juego serán guardadas como pares de coordenadas, con la izquierda superior siendo 0,0 y cada cuadrado del fondo siendo 1 unidad de alto y ancho.

Para interpretar los caracteres en el plano, el constructor del `Nivel` usa el objeto de `caracteresDelNivel`, el cual mapea los elementos a cadenas de caracteres y caracteres de actores a clases. Cuando `tipo` está en la clase actor, su método estático `create` es usado para crear un objeto, el cual es agregado a `iniciarActores` y la función de mapeo regresa `"vacío"` para este cuadrado de fondo.

La posición del actor es guardada como un objeto `Vector`. Este es un vector bidimensional, un objeto con propiedades `x` y `y`, como se vió en los ejercicios de Chapter 6.

Mientras el juego se ejecuta, los actores terminarán en diferentes lugares o incluso desaparecerán completamente (como las monedas lo hacen cuando son

recolectadas). Usaremos una clase `Estado` para dar seguimiento al estado de un juego que en ejecución.

```
class Estado {
  constructor(nivel, actores, estatus) {
    this.nivel = nivel;
    this.actores = actores;
    this.estatus = estatus;
  }

  static start(nivel) {
    return new Estado(nivel, nivel.iniciarActores, "jugando");
  }

  get jugador() {
    return this.actores.find(a => a.tipo == "jugador");
  }
}
```

La propiedad `estatus` cambiará a `"perdido"` or `"ganado"` cuando el juego haya terminado.

Esto es de nuevo una estructura de datos persistente-actualizar el estado del juego crea un nuevo estado y deja el anterior intacto.

Actors

Actor objects represent the current position and state of a given moving element in our game. All actor objects conform to the same interface. Their `pos` property holds the coordinates of the element's top-left corner, and their `size` property holds its size.

Then they have an `update` method, which is used to compute their new state and position after a given time step. It simulates the thing the actor does—moving in response to the arrow keys for the player and bouncing back and forth for the lava—and returns a new, updated actor object.

A `type` property contains a string that identifies the type of the actor—`"player"`, `"coin"`, or `"lava"`. This is useful when drawing the game—the look of the rectangle drawn for an actor is based on its type.

Actor classes have a static `create` method that is used by the `Level` constructor to create an actor from a character in the level plan. It is given the coordinates of the character and the character itself, which is needed because the `Lava` class handles several different characters.

This is the `Vector` class that we'll use for our two-dimensional values, such as the position and size of actors.

```
class Vector {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  plus(other) {
    return new Vector(this.x + other.x, this.y + other.y);
  }
  times(factor) {
    return new Vector(this.x * factor, this.y * factor);
  }
}
```

The `times` method scales a vector by a given number. It will be useful when we need to multiply a speed vector by a time interval to get the distance traveled during that time.

The different types of actors get their own classes since their behavior is very different. Let's define these classes. We'll get to their `update` methods later.

The player class has a property `speed` that stores its current speed to simulate momentum and gravity.

```
class Player {
  constructor(pos, speed) {
    this.pos = pos;
    this.speed = speed;
  }

  get type() { return "player"; }

  static create(pos) {
    return new Player(pos.plus(new Vector(0, -0.5)),
                      new Vector(0, 0));
  }
}

Player.prototype.size = new Vector(0.8, 1.5);
```

Because a player is one-and-a-half squares high, its initial position is set to be half a square above the position where the `@` character appeared. This way, its bottom aligns with the bottom of the square it appeared in.

The `size` property is the same for all instances of `Player`, so we store it on

the prototype rather than on the instances themselves. We could have used a getter like `type`, but that would create and return a new `Vector` object every time the property is read, which would be wasteful. (Strings, being immutable, don't have to be re-created every time they are evaluated.)

When constructing a `Lava` actor, we need to initialize the object differently depending on the character it is based on. Dynamic lava moves along at its current speed until it hits an obstacle. At that point, if it has a `reset` property, it will jump back to its start position (dripping). If it does not, it will invert its speed and continue in the other direction (bouncing).

The `create` method looks at the character that the `Level` constructor passes and creates the appropriate lava actor.

```
class Lava {
  constructor(pos, speed, reset) {
    this.pos = pos;
    this.speed = speed;
    this.reset = reset;
  }

  get type() { return "lava"; }

  static create(pos, ch) {
    if (ch == "=") {
      return new Lava(pos, new Vector(2, 0));
    } else if (ch == "|") {
      return new Lava(pos, new Vector(0, 2));
    } else if (ch == "v") {
      return new Lava(pos, new Vector(0, 3), pos);
    }
  }
}

Lava.prototype.size = new Vector(1, 1);
```

`Coin` actors are relatively simple. They mostly just sit in their place. But to liven up the game a little, they are given a "wobble", a slight vertical back-and-forth motion. To track this, a coin object stores a base position as well as a `wobble` property that tracks the phase of the bouncing motion. Together, these determine the coin's actual position (stored in the `pos` property).

```
class Coin {
  constructor(pos, basePos, wobble) {
    this.pos = pos;
```

```
    this.basePos = basePos;
    this.wobble = wobble;
  }

  get type() { return "coin"; }

  static create(pos) {
    let basePos = pos.plus(new Vector(0.2, 0.1));
    return new Coin(basePos, basePos,
                    Math.random() * Math.PI * 2);
  }
}

Coin.prototype.size = new Vector(0.6, 0.6);
```

In Chapter 13, we saw that `Math.sin` gives us the y-coordinate of a point on a circle. That coordinate goes back and forth in a smooth waveform as we move along the circle, which makes the sine function useful for modeling a wavy motion.

To avoid a situation where all coins move up and down synchronously, the starting phase of each coin is randomized. The *phase* of `Math.sin`'s wave, the width of a wave it produces, is 2π. We multiply the value returned by `Math.random` by that number to give the coin a random starting position on the wave.

We can now define the `levelChars` object that maps plan characters to either background grid types or actor classes.

```
const levelChars = {
  ".": "empty", "#": "wall", "+": "lava",
  "@": Player, "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

That gives us all the parts needed to create a `Level` instance.

```
let simpleLevel = new Level(simpleLevelPlan);
console.log(`${simpleLevel.width} by ${simpleLevel.height}`);
// → 22 by 9
```

The task ahead is to display such levels on the screen and to model time and motion inside them.

Encapsulation as a burden

Most of the code in this chapter does not worry about encapsulation very much for two reasons. First, encapsulation takes extra effort. It makes programs bigger and requires additional concepts and interfaces to be introduced. Since there is only so much code you can throw at a reader before their eyes glaze over, I've made an effort to keep the program small.

Second, the various elements in this game are so closely tied together that if the behavior of one of them changed, it is unlikely that any of the others would be able to stay the same. Interfaces between the elements would end up encoding a lot of assumptions about the way the game works. This makes them a lot less effective—whenever you change one part of the system, you still have to worry about the way it impacts the other parts because their interfaces wouldn't cover the new situation.

Some *cutting points* in a system lend themselves well to separation through rigorous interfaces, but others don't. Trying to encapsulate something that isn't a suitable boundary is a sure way to waste a lot of energy. When you are making this mistake, you'll usually notice that your interfaces are getting awkwardly large and detailed and that they need to be changed often, as the program evolves.

There is one thing that we *will* encapsulate, and that is the drawing subsystem. The reason for this is that we'll display the same game in a different way in the next chapter. By putting the drawing behind an interface, we can load the same game program there and plug in a new display module.

Drawing

The encapsulation of the drawing code is done by defining a *display* object, which displays a given level and state. The display type we define in this chapter is called `DOMDisplay` because it uses DOM elements to show the level.

We'll be using a style sheet to set the actual colors and other fixed properties of the elements that make up the game. It would also be possible to directly assign to the elements' `style` property when we create them, but that would produce more verbose programs.

The following helper function provides a succinct way to create an element and give it some attributes and child nodes:

```
function elt(name, attrs, ...children) {
  let dom = document.createElement(name);
  for (let attr of Object.keys(attrs)) {
```

```
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

A display is created by giving it a parent element to which it should append itself and a level object.

```
class DOMDisplay {
  constructor(parent, level) {
    this.dom = elt("div", {class: "game"}, drawGrid(level));
    this.actorLayer = null;
    parent.appendChild(this.dom);
  }

  clear() { this.dom.remove(); }
}
```

The level's background grid, which never changes, is drawn once. Actors are redrawn every time the display is updated with a given state. The `actorLayer` property will be used to track the element that holds the actors so that they can be easily removed and replaced.

Our coordinates and sizes are tracked in grid units, where a size or distance of 1 means one grid block. When setting pixel sizes, we will have to scale these coordinates up—everything in the game would be ridiculously small at a single pixel per square. The `scale` constant gives the number of pixels that a single unit takes up on the screen.

```
const scale = 20;

function drawGrid(level) {
  return elt("table", {
    class: "background",
    style: `width: ${level.width * scale}px`
  }, ...level.rows.map(row =>
    elt("tr", {style: `height: ${scale}px`},
        ...row.map(type => elt("td", {class: type})))
  ));
}
```

As mentioned, the background is drawn as a `<table>` element. This nicely corresponds to the structure of the `rows` property of the level—each row of the grid is turned into a table row (`<tr>` element). The strings in the grid are used as class names for the table cell (`<td>`) elements. The spread (triple dot) operator is used to pass arrays of child nodes to `elt` as separate arguments.

The following CSS makes the table look like the background we want:

```
.background    { background: rgb(52, 166, 251);
                 table-layout: fixed;
                 border-spacing: 0;              }
.background td { padding: 0;                     }
.lava          { background: rgb(255, 100, 100); }
.wall          { background: white;              }
```

Some of these (`table-layout`, `border-spacing`, and `padding`) are used to suppress unwanted default behavior. We don't want the layout of the table to depend upon the contents of its cells, and we don't want space between the table cells or padding inside them.

The `background` rule sets the background color. CSS allows colors to be specified both as words (`white`) or with a format such as `rgb(R, G, B)`, where the red, green, and blue components of the color are separated into three numbers from 0 to 255. So, in `rgb(52, 166, 251)`, the red component is 52, green is 166, and blue is 251. Since the blue component is the largest, the resulting color will be bluish. You can see that in the `.lava` rule, the first number (red) is the largest.

We draw each actor by creating a DOM element for it and setting that element's position and size based on the actor's properties. The values have to be multiplied by `scale` to go from game units to pixels.

```
function drawActors(actors) {
  return elt("div", {}, ...actors.map(actor => {
    let rect = elt("div", {class: `actor ${actor.type}`});
    rect.style.width = `${actor.size.x * scale}px`;
    rect.style.height = `${actor.size.y * scale}px`;
    rect.style.left = `${actor.pos.x * scale}px`;
    rect.style.top = `${actor.pos.y * scale}px`;
    return rect;
  }));
}
```

To give an element more than one class, we separate the class names by spaces. In the CSS code shown next, the `actor` class gives the actors their absolute position. Their type name is used as an extra class to give them a color. We don't have to define the `lava` class again because we're reusing the class for the lava grid squares we defined earlier.

```
.actor  { position: absolute;           }
.coin   { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64);   }
```

The `syncState` method is used to make the display show a given state. It first removes the old actor graphics, if any, and then redraws the actors in their new positions. It may be tempting to try to reuse the DOM elements for actors, but to make that work, we would need a lot of additional bookkeeping to associate actors with DOM elements and to make sure we remove elements when their actors vanish. Since there will typically be only a handful of actors in the game, redrawing all of them is not expensive.

```
DOMDisplay.prototype.syncState = function(state) {
  if (this.actorLayer) this.actorLayer.remove();
  this.actorLayer = drawActors(state.actors);
  this.dom.appendChild(this.actorLayer);
  this.dom.className = `game ${state.status}`;
  this.scrollPlayerIntoView(state);
};
```

By adding the level's current status as a class name to the wrapper, we can style the player actor slightly differently when the game is won or lost by adding a CSS rule that takes effect only when the player has an ancestor element with a given class.

```
.lost .player {
  background: rgb(160, 64, 64);
}
.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

After touching lava, the player's color turns dark red, suggesting scorching. When the last coin has been collected, we add two blurred white shadows—one to the top left and one to the top right—to create a white halo effect.

We can't assume that the level always fits in the *viewport*—the element into which we draw the game. That is why the `scrollPlayerIntoView` call is needed. It ensures that if the level is protruding outside the viewport, we scroll that viewport to make sure the player is near its center. The following CSS gives the game's wrapping DOM element a maximum size and ensures that anything that sticks out of the element's box is not visible. We also give it a relative position so that the actors inside it are positioned relative to the level's top-left corner.

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

In the `scrollPlayerIntoView` method, we find the player's position and update the wrapping element's scroll position. We change the scroll position by manipulating that element's `scrollLeft` and `scrollTop` properties when the player is too close to the edge.

```
DOMDisplay.prototype.scrollPlayerIntoView = function(state) {
  let width = this.dom.clientWidth;
  let height = this.dom.clientHeight;
  let margin = width / 3;

  // The viewport
  let left = this.dom.scrollLeft, right = left + width;
  let top = this.dom.scrollTop, bottom = top + height;

  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5))
                         .times(scale);

  if (center.x < left + margin) {
    this.dom.scrollLeft = center.x - margin;
  } else if (center.x > right - margin) {
    this.dom.scrollLeft = center.x + margin - width;
  }
  if (center.y < top + margin) {
    this.dom.scrollTop = center.y - margin;
  } else if (center.y > bottom - margin) {
    this.dom.scrollTop = center.y + margin - height;
  }
```

```
};
```

The way the player's center is found shows how the methods on our `Vector` type allow computations with objects to be written in a relatively readable way. To find the actor's center, we add its position (its top-left corner) and half its size. That is the center in level coordinates, but we need it in pixel coordinates, so we then multiply the resulting vector by our display scale.

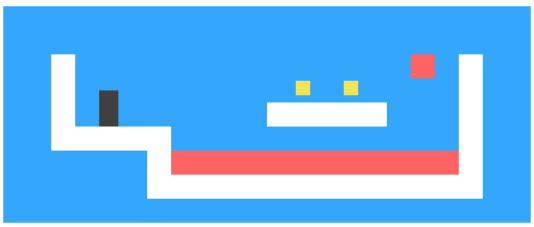
Next, a series of checks verifies that the player position isn't outside of the allowed range. Note that sometimes this will set nonsense scroll coordinates that are below zero or beyond the element's scrollable area. This is okay—the DOM will constrain them to acceptable values. Setting `scrollLeft` to -10 will cause it to become 0.

It would have been slightly simpler to always try to scroll the player to the center of the viewport. But this creates a rather jarring effect. As you are jumping, the view will constantly shift up and down. It is more pleasant to have a "neutral" area in the middle of the screen where you can move around without causing any scrolling.

We are now able to display our tiny level.

```
<link rel="stylesheet" href="css/game.css">

<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLevel);
  display.syncState(State.start(simpleLevel));
</script>
```



The `<link>` tag, when used with `rel="stylesheet"`, is a way to load a CSS file into a page. The file `game.css` contains the styles necessary for our game.

Motion and collision

Now we're at the point where we can start adding motion—the most interesting aspect of the game. The basic approach, taken by most games like this, is to split time into small steps and, for each step, move the actors by a distance corresponding to their speed multiplied by the size of the time step. We'll measure time in seconds, so speeds are expressed in units per second.

Moving things is easy. The difficult part is dealing with the interactions between the elements. When the player hits a wall or floor, they should not simply move through it. The game must notice when a given motion causes an object to hit another object and respond accordingly. For walls, the motion must be stopped. When hitting a coin, it must be collected. When touching lava, the game should be lost.

Solving this for the general case is a big task. You can find libraries, usually called *physics engines*, that simulate interaction between physical objects in two or three dimensions. We'll take a more modest approach in this chapter, handling only collisions between rectangular objects and handling them in a rather simplistic way.

Before moving the player or a block of lava, we test whether the motion would take it inside of a wall. If it does, we simply cancel the motion altogether. The response to such a collision depends on the type of actor—the player will stop, whereas a lava block will bounce back.

This approach requires our time steps to be rather small since it will cause motion to stop before the objects actually touch. If the time steps (and thus the motion steps) are too big, the player would end up hovering a noticeable distance above the ground. Another approach, arguably better but more complicated, would be to find the exact collision spot and move there. We will take the simple approach and hide its problems by ensuring the animation proceeds in small steps.

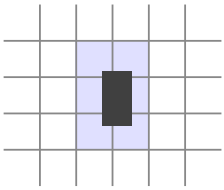
This method tells us whether a rectangle (specified by a position and a size) touches a grid element of the given type.

```
Level.prototype.touches = function(pos, size, type) {
  var xStart = Math.floor(pos.x);
  var xEnd = Math.ceil(pos.x + size.x);
  var yStart = Math.floor(pos.y);
  var yEnd = Math.ceil(pos.y + size.y);

  for (var y = yStart; y < yEnd; y++) {
    for (var x = xStart; x < xEnd; x++) {
      let isOutside = x < 0 || x >= this.width ||
```

```
                    y < 0 || y >= this.height;
      let here = isOutside ? "wall" : this.rows[y][x];
      if (here == type) return true;
    }
  }
  return false;
};
```

The method computes the set of grid squares that the body overlaps with by using `Math.floor` and `Math.ceil` on its coordinates. Remember that grid squares are 1 by 1 units in size. By rounding the sides of a box up and down, we get the range of background squares that the box touches.



We loop over the block of grid squares found by rounding the coordinates and return `true` when a matching square is found. Squares outside of the level are always treated as `"wall"` to ensure that the player can't leave the world and that we won't accidentally try to read outside of the bounds of our `rows` array.

The state `update` method uses `touches` to figure out whether the player is touching lava.

```
State.prototype.update = function(time, keys) {
  let actors = this.actors
    .map(actor => actor.update(time, this, keys));
  let newState = new State(this.level, actors, this.status);

  if (newState.status != "playing") return newState;

  let player = newState.player;
  if (this.level.touches(player.pos, player.size, "lava")) {
    return new State(this.level, actors, "lost");
  }

  for (let actor of actors) {
    if (actor != player && overlap(actor, player)) {
      newState = actor.collide(newState);
    }
  }
```

```
  return newState;
};
```

The method is passed a time step and a data structure that tells it which keys are being held down. The first thing it does is call the `update` method on all actors, producing an array of updated actors. The actors also get the time step, the keys, and the state, so that they can base their update on those. Only the player will actually read keys, since that's the only actor that's controlled by the keyboard.

If the game is already over, no further processing has to be done (the game can't be won after being lost, or vice versa). Otherwise, the method tests whether the player is touching background lava. If so, the game is lost, and we're done. Finally, if the game really is still going on, it sees whether any other actors overlap the player.

Overlap between actors is detected with the `overlap` function. It takes two actor objects and returns true when they touch—which is the case when they overlap both along the x-axis and along the y-axis.

```
function overlap(actor1, actor2) {
  return actor1.pos.x + actor1.size.x > actor2.pos.x &&
         actor1.pos.x < actor2.pos.x + actor2.size.x &&
         actor1.pos.y + actor1.size.y > actor2.pos.y &&
         actor1.pos.y < actor2.pos.y + actor2.size.y;
}
```

If any actor does overlap, its `collide` method gets a chance to update the state. Touching a lava actor sets the game status to `"lost"`. Coins vanish when you touch them and set the status to `"won"` when they are the last coin of the level.

```
Lava.prototype.collide = function(state) {
  return new State(state.level, state.actors, "lost");
};

Coin.prototype.collide = function(state) {
  let filtered = state.actors.filter(a => a != this);
  let status = state.status;
  if (!filtered.some(a => a.type == "coin")) status = "won";
  return new State(state.level, filtered, status);
};
```

Actor updates

Actor objects' `update` methods take as arguments the time step, the state object, and a `keys` object. The one for the `Lava` actor type ignores the `keys` object.

```
Lava.prototype.update = function(time, state) {
  let newPos = this.pos.plus(this.speed.times(time));
  if (!state.level.touches(newPos, this.size, "wall")) {
    return new Lava(newPos, this.speed, this.reset);
  } else if (this.reset) {
    return new Lava(this.reset, this.speed, this.reset);
  } else {
    return new Lava(this.pos, this.speed.times(-1));
  }
};
```

This `update` method computes a new position by adding the product of the time step and the current speed to its old position. If no obstacle blocks that new position, it moves there. If there is an obstacle, the behavior depends on the type of the lava block—dripping lava has a `reset` position, to which it jumps back when it hits something. Bouncing lava inverts its speed by multiplying it by -1 so that it starts moving in the opposite direction.

Coins use their `update` method to wobble. They ignore collisions with the grid since they are simply wobbling around inside of their own square.

```
const wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.update = function(time) {
  let wobble = this.wobble + time * wobbleSpeed;
  let wobblePos = Math.sin(wobble) * wobbleDist;
  return new Coin(this.basePos.plus(new Vector(0, wobblePos)),
                  this.basePos, wobble);
};
```

The `wobble` property is incremented to track time and then used as an argument to `Math.sin` to find the new position on the wave. The coin's current position is then computed from its base position and an offset based on this wave.

That leaves the player itself. Player motion is handled separately per axis because hitting the floor should not prevent horizontal motion, and hitting a wall should not stop falling or jumping motion.

```
const playerXSpeed = 7;
const gravity = 30;
const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {
  let xSpeed = 0;
  if (keys.ArrowLeft) xSpeed -= playerXSpeed;
  if (keys.ArrowRight) xSpeed += playerXSpeed;
  let pos = this.pos;
  let movedX = pos.plus(new Vector(xSpeed * time, 0));
  if (!state.level.touches(movedX, this.size, "wall")) {
    pos = movedX;
  }

  let ySpeed = this.speed.y + time * gravity;
  let movedY = pos.plus(new Vector(0, ySpeed * time));
  if (!state.level.touches(movedY, this.size, "wall")) {
    pos = movedY;
  } else if (keys.ArrowUp && ySpeed > 0) {
    ySpeed = -jumpSpeed;
  } else {
    ySpeed = 0;
  }
  return new Player(pos, new Vector(xSpeed, ySpeed));
};
```

The horizontal motion is computed based on the state of the left and right arrow keys. When there's no wall blocking the new position created by this motion, it is used. Otherwise, the old position is kept.

Vertical motion works in a similar way but has to simulate jumping and gravity. The player's vertical speed (`ySpeed`) is first accelerated to account for gravity.

We check for walls again. If we don't hit any, the new position is used. If there *is* a wall, there are two possible outcomes. When the up arrow is pressed *and* we are moving down (meaning the thing we hit is below us), the speed is set to a relatively large, negative value. This causes the player to jump. If that is not the case, the player simply bumped into something, and the speed is set to zero.

The gravity strength, jumping speed, and pretty much all other constants in this game have been set by trial and error. I tested values until I found a combination I liked.

Tracking keys

For a game like this, we do not want keys to take effect once per keypress. Rather, we want their effect (moving the player figure) to stay active as long as they are held.

We need to set up a key handler that stores the current state of the left, right, and up arrow keys. We will also want to call `preventDefault` for those keys so that they don't end up scrolling the page.

The following function, when given an array of key names, will return an object that tracks the current position of those keys. It registers event handlers for `"keydown"` and `"keyup"` events and, when the key code in the event is present in the set of codes that it is tracking, updates the object.

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys =
  trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);
```

The same handler function is used for both event types. It looks at the event object's `type` property to determine whether the key state should be updated to true (`"keydown"`) or false (`"keyup"`).

Running the game

The `requestAnimationFrame` function, which we saw in Chapter 13, provides a good way to animate a game. But its interface is quite primitive—using it requires us to track the time at which our function was called the last time around and call `requestAnimationFrame` again after every frame.

Let's define a helper function that wraps those boring parts in a convenient interface and allows us to simply call `runAnimation`, giving it a function that

expects a time difference as an argument and draws a single frame. When the frame function returns the value `false`, the animation stops.

```
function runAnimation(frameFunc) {
  let lastTime = null;
  function frame(time) {
    if (lastTime != null) {
      let timeStep = Math.min(time - lastTime, 100) / 1000;
      if (frameFunc(timeStep) === false) return;
    }
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);
}
```

I have set a maximum frame step of 100 milliseconds (one-tenth of a second). When the browser tab or window with our page is hidden, `requestAnimationFrame` calls will be suspended until the tab or window is shown again. In this case, the difference between `lastTime` and `time` will be the entire time in which the page was hidden. Advancing the game by that much in a single step would look silly and might cause weird side effects, such as the player falling through the floor.

The function also converts the time steps to seconds, which are an easier quantity to think about than milliseconds.

The `runLevel` function takes a `Level` object and a display constructor and returns a promise. It displays the level (in `document.body`) and lets the user play through it. When the level is finished (lost or won), `runLevel` waits one more second (to let the user see what happens) and then clears the display, stops the animation, and resolves the promise to the game's end status.

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.syncState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
```

```
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

A game is a sequence of levels. Whenever the player dies, the current level is restarted. When a level is completed, we move on to the next level. This can be expressed by the following function, which takes an array of level plans (strings) and a display constructor:

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level]),
                                Display);
    if (status == "won") level++;
  }
  console.log("You've won!");
}
```

Because we made `runLevel` return a promise, `runGame` can be written using an `async` function, as shown in Chapter 11. It returns another promise, which resolves when the player finishes the game.

There is a set of level plans available in the `GAME_LEVELS` binding in this chapter's sandbox (*https://eloquentjavascript.net/code#16*). This page feeds them to `runGame`, starting an actual game.

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

Exercises

Game over

It's traditional for platform games to have the player start with a limited number of *lives* and subtract one life each time they die. When the player is out of lives, the game restarts from the beginning.

Adjust `runGame` to implement lives. Have the player start with three. Output the current number of lives (using `console.log`) every time a level starts.

Pausing the game

Make it possible to pause (suspend) and unpause the game by pressing the Esc key.

This can be done by changing the `runLevel` function to use another keyboard event handler and interrupting or resuming the animation whenever the Esc key is hit.

The `runAnimation` interface may not look like it is suitable for this at first glance, but it is if you rearrange the way `runLevel` calls it.

When you have that working, there is something else you could try. The way we have been registering keyboard event handlers is somewhat problematic. The `arrowKeys` object is currently a global binding, and its event handlers are kept around even when no game is running. You could say they *leak* out of our system. Extend `trackKeys` to provide a way to unregister its handlers and then change `runLevel` to register its handlers when it starts and unregister them again when it is finished.

A monster

It is traditional for platform games to have enemies that you can jump on top of to defeat. This exercise asks you to add such an actor type to the game.

We'll call it a monster. Monsters move only horizontally. You can make them move in the direction of the player, bounce back and forth like horizontal lava, or have any movement pattern you want. The class doesn't have to handle falling, but it should make sure the monster doesn't walk through walls.

When a monster touches the player, the effect depends on whether the player is jumping on top of them or not. You can approximate this by checking whether the player's bottom is near the monster's top. If this is the case, the monster disappears. If not, the game is lost.

> *"Un estudiante preguntó: 'Los programadores de antaño sólo usaban máquinas simples y ningún lenguaje de programación, pero hicieron programas hermosos. ¿Por qué usamos máquinas y lenguajes de programación complicados?'. Fu-Tzu respondió, Los constructores de antaño sólo usaban palos y arcilla, pero hicieron bellas chozas."'*
>
> —Master Yuan-Ma, The Book of Programming

Chapter 16

Node.js

Hasta ahora, hemos usado el lenguaje JavaScript en un solo entorno: el navegador. Este capítulo y el siguiente presentarán brevemente Node.js, un programa que te permite aplicar tus habilidades de JavaScript fuera del navegador. Con él, puedes crear cualquier cosa, desde pequeñas herramientas de línea de comandos hasta servidores HTTP que potencian los sitios web dinámicos.

Estos capítulos tienen como objetivo enseñarte los principales conceptos que Node.js utiliza y darte suficiente información para escribir programas útiles de este. No intentan ser un tratamiento completo, ni siquiera minucioso, de la plataforma.

Si quieres seguir y ejecutar el código de este capítulo, necesitarás instalar Node.js versión 10.1 o superior. Para ello, dirigete a *https://nodejs.org* y sigue las instrucciones de instalación para tu sistema operativo. También podrás encontrar más documentación para Node.js allí.

Antecedentes

Uno de los problemas más difíciles de los sistemas de escritura que se comunican a través de la red es la gestión de entrada y salida, es decir, la lectura y la escritura de datos hacia y desde la red y el disco duro. Mover los datos toma tiempo, y programarlos con habilidad puede hacer una gran diferencia en que tan rápido un sistema responde al usuario o a las peticiones de red.

En tales programas, la programación asíncrona suele ser útil. Permite que el programa envíe y reciba datos desde y hacia múltiples dispositivos al mismo tiempo sin necesidad de una complicada administración y sincronización de hilos.

Node fue concebido inicialmente con el propósito de hacer la programación asíncrona fácil y conveniente. JavaScript se presta bien a un sistema como Node. Es uno de los pocos lenguajes de programación que no tiene una forma incorporada de hacer entrada- y salida. Por lo tanto, JavaScript podría encajar en un enfoque bastante excéntrico de Node para hacer entrada y salida sin

terminar con dos interfaces inconsistentes. En 2009, cuando se diseñó Node, la gente ya estaba haciendo programación basada en llamadas en el navegador, por lo que la comunidad en torno al lenguaje estaba acostumbrada a un estilo de programación asíncrono.

El comando Node

Cuando Node.js se instala en un sistema, proporciona un programa llamado `node`, que se utiliza para ejecutar archivos JavaScript. Digamos que tienes un archivo `hello.js`, que contiene este código:

```
let message = "Hello world";
console.log(message);
```

Entonces podrás lanzar `node` desde la línea de comandos de esta manera para ejecutar el programa:

```
$ node hello.js
Hello world
```

El método `console.log` en Node hace algo similar a lo que realiza en el navegador. Imprime una porción de texto. Pero en Node, el texto irá al flujo de la salida estándar del proceso, en lugar de la consola de JavaScript del navegador. Cuando ejecutas Node desde la línea de comandos, eso significa que verás los valores escritos en tu terminal.

Si ejecutas `node` sin indicar un archivo, te proporciona un prompt en el cual podrás escribir código JavaScript e inmediatamente ver el resultado.

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

La referencia `process`, al igual que la referencia `console`, está disponible globalmente en Node. Esta proporciona varias formas de inspeccionar y manipular el programa actual. El método `exit` finaliza el proceso y se le puede pasar un código de estado de salida, esto le dice al programa que inició node (en este

caso, el shell de la línea de comandos) si el programa finalizó con éxito (código cero) o si se encontró con un error (cualquier otro código).

Para obtener los argumentos de la línea de comandos proporcionados a tu script, puede leer `process.argv`, que es un conjunto de cadenas. Ten en cuenta que también incluye el nombre del comando `node` y el nombre de tu script, por lo que los argumentos reales comienzan en el índice 2. Si `showargv.js` contiene la sentencia console.log(process.argv), podrías ejecutarlo así:

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

Todas las referencias globales JavaScript estándar, como Array, Math y JSON, también están presentes en el entorno de Node. La funcionalidad relacionada con el navegador, como el documento o el prompt, no lo son.

Módulos

Más allá de las referencias que mencioné, como `console` y `process`, Node pone unas cuantas referencias adicionales en el ámbito global. Si quieres acceder a la funcionalidad integrada, tienes que pedírsela al sistema de módulos.

El sistema de módulos CommonJS, basado en la función `require`, se describió en el Capítulo 10. Este sistema está incorporado en Node y se usa para cargar cualquier cosa desde módulos incorporados hasta paquetes descargados y archivos que forman parte de tu propio programa.

Cuando se llama a `require`, Node tiene que resolver la cadena dada a un archivo real que pueda cargar. Las rutas que empiezan por `/`, `./`, o `../` se resuelven relativamente a la ruta del módulo actual, donde `.` se refiere al directorio actual, `../` a un directorio superior, y `/` a la raíz del sistema de archivos. Así que si pide `"./graph"` del archivo `/tmp/robot/robot.js`, Node intentará cargar el archivo `/tmp/robot/graph.js`.

La extension `.js` puede ser omitida, y Node la añadirá si existe tal archivo. Si la ruta requerida se refiere a un directorio, Node intentará cargar el archivo llamado `index.js` en ese directorio.

Cuando una cadena que no parece una ruta relativa o absoluta es dada a `require`, se asume que se refiere a un módulo incorporado o a un módulo instalado en un directorio `node_modules`. Por ejemplo, `require("fs")` te dará el módulo incorporado del sistema de archivos de Node. Y `require("robot")` podría intentar cargar la biblioteca que se encuentra en `node_modules/robot/`. Una forma común de instalar tales librerías es usando NPM, al cual volveremos

en un momento.

Vamos a crear un pequeño proyecto que consiste en dos archivos. El primero, llamado `main.js`, define un script que puede ser llamado desde la línea de comandos para invertir una cadena.

```
const {reverse} = require("./reverse");

// El índice 2 contiene el primer argumento de la línea de comandos
let argument = process.argv[2];

console.log(reverse(argument));
```

El archivo `reverse.js` define una biblioteca para invertir cadenas, el cual puede ser utilizado tanto por la herramienta de línea de comandos como por otros scripts que necesiten acceso directo a una función de inversión de cadenas.

```
exports.reverse = function(string) {
  return Array.from(string).reverse().join("");
};
```

Recuerde que al añadir propiedades a `exports` estas se agregan a la interfaz del módulo. Dado que Node.js trata a los archivos como módulos CommonJS, `main.js` puede tomar la función exportada `reverse` desde `reverse.js`.

Ahora podemos llamar a nuestra herramienta así:

```
$ node main.js JavaScript
tpircSavaJ
```

Instalación con NPM

NPM, el cual fue introducido en el Capítulo 10, es un repositorio en línea de módulos JavaScript, muchos de los cuales están escritos específicamente para Node. Cuando instalas Node en tu ordenador, también obtienes el comando `npm`, que puedes usar para interactuar con este repositorio.

El uso principal de NPM es descargar paquetes. Vimos el paquete `ini` en el Capítulo 10. nosotros podemos usar NPM para buscar e instalar ese paquete en nuestro ordenador.

```
$ npm install ini
npm WARN enoent ENOENT: no such file or directory,
```

```
         open '/tmp/package.json'
+ ini@1.3.5
added 1 package in 0.552s

$ node
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

Después de ejecutar `npm install`, NPM habrá creado un directorio llamado `node_modules`. Dentro de ese directorio habrá un directorio `ini` que contiene la biblioteca. Puedes abrirlo y mirar el código. Cuando llamamos a `require("ini")`, esta librería se carga, y podemos llamar a la propiedad `parse` para analizar un archivo de configuración.

Por defecto, NPM instala los paquetes bajo el directorio actual, en lugar de una ubicación central. Si está acostumbrado a otros gestores de paquetes, esto puede parecer inusual, pero tiene ventajas, pone a cada aplicación en control total de los paquetes que instala y facilita la gestión de versiones y la limpieza al eliminar una aplicación.

Archivos de paquetes

En el ejemplo `npm install`, se podía ver una advertencia sobre el hecho de que el archivo `package.json` no existía. Se recomienda crear dicho archivo para cada proyecto, ya sea manualmente o ejecutando `npm init`. Este contiene alguna información sobre el proyecto, como lo son su nombre y versión, y enumera sus dependencias.

La simulación del robot del Capítulo 7, tal como fue modularizada en el ejercicio del Capítulo 10, podría tener un archivo `package.json` como este:

```
{
  "author": "Marijn Haverbeke",
  "name": "eloquent-javascript-robot",
  "description": "Simulation of a package-delivery robot",
  "version": "1.0.0",
  "main": "run.js",
  "dependencies": {
    "dijkstrajs": "^1.0.1",
    "random-item": "^1.0.0"
  },
  "license": "ISC"
}
```

Cuando ejecutas `npm install` sin nombrar un paquete a instalar, NPM instalará las dependencias listadas en `package.json`. Cuando instale un paquete específico que no esté ya listado como una dependencia, NPM lo añadirá al `package.json`.

Versiones

Un archivo `package.json` lista tanto la versión propia del programa como las versiones de sus dependencias. Las versiones son una forma de lidiar con el hecho de que los paquetes evolucionan por separado, y el código escrito para trabajar con un paquete tal y como existía en un determinado momento pueda no funcionar con una versión posterior y modificada del paquete.

NPM exige que sus paquetes sigan un esquema llamado *versionado semántico*, el cual codifica cierta información sobre qué versiones son *compatibles* (no rompan la vieja interfaz) en el número de versión. Una versión semántica consiste en tres números, separados por puntos, como lo es `2.3.0`. Cada vez que una nueva funcionalidad es agregada, el número intermedio tiene que ser incrementado. Cada vez que se rompe la compatibilidad, de modo que el código existente que utiliza el paquete podría no funcionar con esa nueva versión, el primer número tiene que ser incrementado.

Un carácter cuña (`^`) delante del número de versión para una dependencia en el `package.json` indica que cualquier versión compatible con el número dado puede ser instalada. Así, por ejemplo, `"^2.3.0"` significaría que cualquier versión mayor o igual a 2.3.0 y menor a 3.0.0 está permitida.

El comando `npm` también se usa para publicar nuevos paquetes o nuevas versiones de paquetes. Si ejecutas `npm publish` en un directorio que tiene un archivo `package.json`, esto publicará un paquete con el nombre y la versión que aparece en el archivo JSON. Cualquiera puede publicar paquetes en NPM, aunque sólo bajo un nombre de paquete que no esté en uso todavía, ya que sería algo aterrador si personas al azar pudieran actualizar los paquetes existentes.

Dado que el programa `npm` es una pieza de software que habla con un sistema abierto—El registro de paquetes— no es el único que lo hace. Otro programa, `yarn`, el cual puede ser instalado desde el registro NPM, cumple el mismo papel que `npm` usando una interfaz y una estrategia de instalación algo diferente.

Este libro no profundizará en los detalles del uso del NPM. Consulte *https://npmjs.org* para obtener más documentación y formas para buscar paquetes.

El módulo del sistema de archivos

Uno de los módulos incorporados más utilizados en Node es el módulo `fs`, que significa *sistema de archivos*. Este exporta funciones para trabajar con archivos y directorios.

Por ejemplo, la función llamada `readFile` lee un archivo y luego llama un callback con el contenido del archivo.

```
let {readFile} = require("fs");
readFile("file.txt", "utf8", (error, text) => {
  if (error) throw error;
  console.log("The file contains:", text);
});
```

El segundo argumento para `readFile` indica la *codificación de caracteres* utilizada para decodificar el archivo en una cadena. Hay varias formas de codificar texto en datos binarios, pero la mayoría de los sistemas modernos utilizan UTF-8. Por lo tanto, a menos que tenga razones para creer que se utiliza otra codificación, pase `"utf8"` cuando lea un archivo de texto. Si no pasas una codificación, Node asumirá que estás interesado en los datos binarios y te dará un objeto `Buffer` en lugar de una cadena. Este es un objeto tipo array que contiene números que representan los bytes (trozos de 8 bits de datos) en los archivos.

```
const {readFile} = require("fs");
readFile("file.txt", (error, buffer) => {
  if (error) throw error;
  console.log("The file contained", buffer.length, "bytes.",
              "The first byte is:", buffer[0]);
});
```

Una función similar, `writeFile`, es usada para escribir un archivo en el disco.

```
const {writeFile} = require("fs");
writeFile("graffiti.txt", "Node was here", err => {
  if (err) console.log(`Failed to write file: ${err}`);
  else console.log("File written.");
});
```

Aquí no fue necesario especificar la codificación—`writeFile` asumirá que cuando se le da una cadena a escribir, en lugar de un objeto `Buffer`, debe

escribirlo como texto usando su codificación de caracteres por defecto, que es UTF-8.

El módulo `fs` contiene muchas otras funciones útiles: `readdir` devolverá los archivos de un directorio como un array de cadenas, `stat` recuperará información sobre un archivo, `rename` renombrará un archivo, `unlink` eliminará uno, y así sucesivamente. Ve la documentación en *https://nodejs.org* para más detalle.

La mayoría de estos toman una función de devolución como último parámetro, el cual lo llaman ya sea con un error (el primer argumento) o con un resultado exitoso (el segundo). Como vimos en el Capítulo 11, este estilo de programación tiene sus desventajas , la mayor de las cuales es que el manejo de los errores se vuelve enredado y propenso a errores.

Aunque las promesas han sido parte de JavaScript por un tiempo, hasta el momento de escribir este documento su integración en Node.js es aun un trabajo en progreso. Hay un objeto `promises` exportado del paquete `fs` desde la versión 10.1 que contiene la mayoría de las mismas funciones que `fs` pero que utiliza promesas en lugar de funciones de devolución de llamada.

```
const {readFile} = require("fs").promises;
readFile("file.txt", "utf8")
  .then(text => console.log("The file contains:", text));
```

Algunas veces no se necesita asincronicidad, y esto sólo obstaculiza el camino. Muchas de las funciones en `fs` también tienen una variante síncrona, que tiene el mismo nombre con `Sync` añadido al final. Por ejemplo, la versión síncrona de `readFile` es llamada `readFileSync`.

```
const {readFileSync} = require("fs");
console.log("The file contains:",
            readFileSync("file.txt", "utf8"));
```

Ten en cuenta que mientras se realiza esta operación sincrónica, tu programa se detiene por completo. Si debe responder al usuario o a otras máquinas en la red, quedarse atascado en una acción síncrona puede producir retrasos molestos.

The HTTP module

Another central module is called `http`. It provides functionality for running HTTP servers and making HTTP requests.

This is all it takes to start an HTTP server:

```
const {createServer} = require("http");
let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

If you run this script on your own machine, you can point your web browser at *http://localhost:8000/hello* to make a request to your server. It will respond with a small HTML page.

The function passed as argument to `createServer` is called every time a client connects to the server. The `request` and `response` bindings are objects representing the incoming and outgoing data. The first contains information about the request, such as its `url` property, which tells us to what URL the request was made.

So, when you open that page in your browser, it sends a request to your own computer. This causes the server function to run and send back a response, which you can then see in the browser.

To send something back, you call methods on the `response` object. The first, `writeHead`, will write out the response headers (see Chapter ?). You give it the status code (200 for "OK" in this case) and an object that contains header values. The example sets the `Content-Type` header to inform the client that we'll be sending back an HTML document.

Next, the actual response body (the document itself) is sent with `response.write`. You are allowed to call this method multiple times if you want to send the response piece by piece, for example to stream data to the client as it becomes available. Finally, `response.end` signals the end of the response.

The call to `server.listen` causes the server to start waiting for connections on port 8000. This is why you have to connect to *localhost:8000* to speak to this server, rather than just *localhost*, which would use the default port 80.

When you run this script, the process just sits there and waits. When a script is listening for events—in this case, network connections—`node` will not automatically exit when it reaches the end of the script. To close it, press CONTROL-C.

A real web server usually does more than the one in the example—it looks at the request's method (the `method` property) to see what action the client is

trying to perform and looks at the request's URL to find out which resource this action is being performed on. We'll see a more advanced server later in this chapter.

To act as an HTTP *client*, we can use the `request` function in the `http` module.

```
const {request} = require("http");
let requestStream = request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, response => {
  console.log("Server responded with status code",
              response.statusCode);
});
requestStream.end();
```

The first argument to `request` configures the request, telling Node what server to talk to, what path to request from that server, which method to use, and so on. The second argument is the function that should be called when a response comes in. It is given an object that allows us to inspect the response, for example to find out its status code.

Just like the `response` object we saw in the server, the object returned by `request` allows us to stream data into the request with the `write` method and finish the request with the `end` method. The example does not use `write` because `GET` requests should not contain data in their request body.

There's a similar `request` function in the `https` module that can be used to make requests to `https:` URLs.

Making requests with Node's raw functionality is rather verbose. There are much more convenient wrapper packages available on NPM. For example, `node-fetch` provides the promise-based `fetch` interface that we know from the browser.

Streams

We have seen two instances of writable streams in the HTTP examples—namely, the response object that the server could write to and the request object that was returned from `request`.

Writable streams are a widely used concept in Node. Such objects have a `write` method that can be passed a string or a `Buffer` object to write something

to the stream. Their `end` method closes the stream and optionally takes a value to write to the stream before closing. Both of these methods can also be given a callback as an additional argument, which they will call when the writing or closing has finished.

It is possible to create a writable stream that points at a file with the `createWriteStream` function from the `fs` module. Then you can use the `write` method on the resulting object to write the file one piece at a time, rather than in one shot as with `writeFile`.

Readable streams are a little more involved. Both the `request` binding that was passed to the HTTP server's callback and the `response` binding passed to the HTTP client's callback are readable streams—a server reads requests and then writes responses, whereas a client first writes a request and then reads a response. Reading from a stream is done using event handlers, rather than methods.

Objects that emit events in Node have a method called `on` that is similar to the `addEventListener` method in the browser. You give it an event name and then a function, and it will register that function to be called whenever the given event occurs.

Readable streams have `"data"` and `"end"` events. The first is fired every time data comes in, and the second is called whenever the stream is at its end. This model is most suited for *streaming* data that can be immediately processed, even when the whole document isn't available yet. A file can be read as a readable stream by using the `createReadStream` function from `fs`.

This code creates a server that reads request bodies and streams them back to the client as all-uppercase text:

```
const {createServer} = require("http");
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
}).listen(8000);
```

The `chunk` value passed to the data handler will be a binary `Buffer`. We can convert this to a string by decoding it as UTF-8 encoded characters with its `toString` method.

The following piece of code, when run with the uppercasing server active, will send a request to that server and write out the response it gets:

```
const {request} = require("http");
```

```
request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, response => {
  response.on("data", chunk =>
    process.stdout.write(chunk.toString()));
}).end("Hello server");
// → HELLO SERVER
```

The example writes to `process.stdout` (the process's standard output, which is a writable stream) instead of using `console.log`. We can't use `console.log` because it adds an extra newline character after each piece of text that it writes, which isn't appropriate here since the response may come in as multiple chunks.

A FILE SERVER

Let's combine our newfound knowledge about HTTP servers and working with the file system to create a bridge between the two: an HTTP server that allows remote access to a file system. Such a server has all kinds of uses—it allows web applications to store and share data, or it can give a group of people shared access to a bunch of files.

When we treat files as HTTP resources, the HTTP methods `GET`, `PUT`, and `DELETE` can be used to read, write, and delete the files, respectively. We will interpret the path in the request as the path of the file that the request refers to.

We probably don't want to share our whole file system, so we'll interpret these paths as starting in the server's working directory, which is the directory in which it was started. If I ran the server from `/tmp/public/` (or `C:\tmp\public\` on Windows), then a request for `/file.txt` should refer to `/tmp/public/file.txt` (or `C:\tmp\public\file.txt`).

We'll build the program piece by piece, using an object called `methods` to store the functions that handle the various HTTP methods. Method handlers are `async` functions that get the request object as argument and return a promise that resolves to an object that describes the response.

```
const {createServer} = require("http");

const methods = Object.create(null);
```

```
createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
       response.writeHead(status, {"Content-Type": type});
       if (body && body.pipe) body.pipe(response);
       else response.end(body);
    });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

This starts a server that just returns 405 error responses, which is the code used to indicate that the server refuses to handle a given method.

When a request handler's promise is rejected, the `catch` call translates the error into a response object, if it isn't one already, so that the server can send back an error response to inform the client that it failed to handle the request.

The `status` field of the response description may be omitted, in which case it defaults to 200 (OK). The content type, in the `type` property, can also be left off, in which case the response is assumed to be plain text.

When the value of `body` is a readable stream, it will have a `pipe` method that is used to forward all content from a readable stream to a writable stream. If not, it is assumed to be either `null` (no body), a string, or a buffer, and it is passed directly to the response's `end` method.

To figure out which file path corresponds to a request URL, the `urlPath` function uses Node's built-in `url` module to parse the URL. It takes its pathname, which will be something like `"/file.txt"`, decodes that to get rid of the `%20`-style escape codes, and resolves it relative to the program's working directory.

```
const {parse} = require("url");
const {resolve, sep} = require("path");
```

```
const baseDirectory = process.cwd();

function urlPath(url) {
  let {pathname} = parse(url);
  let path = resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

As soon as you set up a program to accept network requests, you have to start worrying about security. In this case, if we aren't careful, it is likely that we'll accidentally expose our whole file system to the network.

File paths are strings in Node. To map such a string to an actual file, there is a nontrivial amount of interpretation going on. Paths may, for example, include `../` to refer to a parent directory. So one obvious source of problems would be requests for paths like `/../secret_file`.

To avoid such problems, `urlPath` uses the `resolve` function from the `path` module, which resolves relative paths. It then verifies that the result is *below* the working directory. The `process.cwd` function (where `cwd` stands for "current working directory") can be used to find this working directory. The `sep` binding from the `path` package is the system's path separator—a backslash on Windows and a forward slash on most other systems. When the path doesn't start with the base directory, the function throws an error response object, using the HTTP status code indicating that access to the resource is forbidden.

We'll set up the `GET` method to return a list of files when reading a directory and to return the file's content when reading a regular file.

One tricky question is what kind of `Content-Type` header we should set when returning a file's content. Since these files could be anything, our server can't simply return the same content type for all of them. NPM can help us again here. The `mime` package (content type indicators like `text/plain` are also called *MIME types*) knows the correct type for a large number of file extensions.

The following `npm` command, in the directory where the server script lives, installs a specific version of `mime`:

```
$ npm install mime@2.2.0
```

When a requested file does not exist, the correct HTTP status code to return

is 404. We'll use the `stat` function, which looks up information about a file, to find out both whether the file exists and whether it is a directory.

```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");

methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: mime.getType(path)};
  }
};
```

Because it has to touch the disk and thus might take a while, `stat` is asynchronous. Since we're using promises rather than callback style, it has to be imported from `promises` instead of directly from `fs`.

When the file does not exist, `stat` will throw an error object with a `code` property of `"ENOENT"`. These somewhat obscure, Unix-inspired codes are how you recognize error types in Node.

The `stats` object returned by `stat` tells us a number of things about a file, such as its size (`size` property) and its modification date (`mtime` property). Here we are interested in the question of whether it is a directory or a regular file, which the `isDirectory` method tells us.

We use `readdir` to read the array of files in a directory and return it to the client. For normal files, we create a readable stream with `createReadStream` and return that as the body, along with the content type that the `mime` package gives us for the file's name.

The code to handle `DELETE` requests is slightly simpler.

```
const {rmdir, unlink} = require("fs").promises;

methods.DELETE = async function(request) {
```

```
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 204};
  }
  if (stats.isDirectory()) await rmdir(path);
  else await unlink(path);
  return {status: 204};
};
```

When an HTTP response does not contain any data, the status code 204 ("no content") can be used to indicate this. Since the response to deletion doesn't need to transmit any information beyond whether the operation succeeded, that is a sensible thing to return here.

You may be wondering why trying to delete a nonexistent file returns a success status code, rather than an error. When the file that is being deleted is not there, you could say that the request's objective is already fulfilled. The HTTP standard encourages us to make requests *idempotent*, which means that making the same request multiple times produces the same result as making it once. In a way, if you try to delete something that's already gone, the effect you were trying to do has been achieved—the thing is no longer there.

This is the handler for `PUT` requests:

```
const {createWriteStream} = require("fs");

function pipeStream(from, to) {
  return new Promise((resolve, reject) => {
    from.on("error", reject);
    to.on("error", reject);
    to.on("finish", resolve);
    from.pipe(to);
  });
}

methods.PUT = async function(request) {
  let path = urlPath(request.url);
  await pipeStream(request, createWriteStream(path));
  return {status: 204};
};
```

We don't need to check whether the file exists this time—if it does, we'll just overwrite it. We again use `pipe` to move data from a readable stream to a writable one, in this case from the request to the file. But since `pipe` isn't written to return a promise, we have to write a wrapper, `pipeStream`, that creates a promise around the outcome of calling `pipe`.

When something goes wrong when opening the file, `createWriteStream` will still return a stream, but that stream will fire an `"error"` event. The output stream to the request may also fail, for example if the network goes down. So we wire up both streams' `"error"` events to reject the promise. When `pipe` is done, it will close the output stream, which causes it to fire a `"finish"` event. That's the point where we can successfully resolve the promise (returning nothing).

The full script for the server is available at *https://eloquentjavascript.net/code/file_server.js*. You can download that and, after installing its dependencies, run it with Node to start your own file server. And, of course, you can modify and extend it to solve this chapter's exercises or to experiment.

The command line tool `curl`, widely available on Unix-like systems (such as macOS and Linux), can be used to make HTTP requests. The following session briefly tests our server. The `-X` option is used to set the request's method, and `-d` is used to include a request body.

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

The first request for `file.txt` fails since the file does not exist yet. The `PUT` request creates the file, and behold, the next request successfully retrieves it. After deleting it with a `DELETE` request, the file is again missing.

Summary

Node is a nice, small system that lets us run JavaScript in a nonbrowser context. It was originally designed for network tasks to play the role of a *node* in a network. But it lends itself to all kinds of scripting tasks, and if writing JavaScript is something you enjoy, automating tasks with Node works well.

NPM provides packages for everything you can think of (and quite a few things you'd probably never think of), and it allows you to fetch and install those packages with the `npm` program. Node comes with a number of built-in modules, including the `fs` module for working with the file system and the `http` module for running HTTP servers and making HTTP requests.

All input and output in Node is done asynchronously, unless you explicitly use a synchronous variant of a function, such as `readFileSync`. When calling such asynchronous functions, you provide callback functions, and Node will call them with an error value and (if available) a result when it is ready.

Exercises

Search tool

On Unix systems, there is a command line tool called `grep` that can be used to quickly search files for a regular expression.

Write a Node script that can be run from the command line and acts somewhat like `grep`. It treats its first command line argument as a regular expression and treats any further arguments as files to search. It should output the names of any file whose content matches the regular expression.

When that works, extend it so that when one of the arguments is a directory, it searches through all files in that directory and its subdirectories.

Use asynchronous or synchronous file system functions as you see fit. Setting things up so that multiple asynchronous actions are requested at the same time might speed things up a little, but not a huge amount, since most file systems can read only one thing at a time.

Directory creation

Though the `DELETE` method in our file server is able to delete directories (using `rmdir`), the server currently does not provide any way to *create* a directory.

Add support for the `MKCOL` method ("make collection"), which should create a directory by calling `mkdir` from the `fs` module. `MKCOL` is not a widely used HTTP method, but it does exist for this same purpose in the *WebDAV* standard, which specifies a set of conventions on top of HTTP that make it suitable for creating documents.

Since the file server serves up any kind of file and even includes the right `Content-Type` header, you can use it to serve a website. Since it allows everybody to delete and replace files, it would be an interesting kind of website: one that can be modified, improved, and vandalized by everybody who takes the time to create the right HTTP request.

Write a basic HTML page that includes a simple JavaScript file. Put the files in a directory served by the file server and open them in your browser.

Next, as an advanced exercise or even a weekend project, combine all the knowledge you gained from this book to build a more user-friendly interface for modifying the website—from *inside* the website.

Use an HTML form to edit the content of the files that make up the website, allowing the user to update them on the server by using HTTP requests, as described in Chapter ?.

Start by making only a single file editable. Then make it so that the user can select which file to edit. Use the fact that our file server returns lists of files when reading a directory.

Don't work directly in the code exposed by the file server since if you make a mistake, you are likely to damage the files there. Instead, keep your work outside of the publicly accessible directory and copy it there when testing.

"Si tienes conocimiento, deja a otros encender sus velas allí."

—Margaret Fuller

Chapter 17

Proyecto: Sitio web para compartir habilidades

Una reunión para *compartir habilidades* es un evento en donde personas con intereses comunes se juntan para dar pequeñas presentaciones informales acerca de cosas que saben. En un reunión de jardinería alguien explicaría como cultivar apio. O en un grupo de programación, podrías presentarte y hablarle a las personas acerca de Node.js.

En estas reuniones, también llamadas *grupos de usuarios* cuando se tratan de computadoras, son una forma genial de ampliar tus horizontes, aprender acerca de nuevos desarrollos, o simplemente conoce gente con intereses similares. Muchas ciudades grandes tienen reuniones sobre JavaScript. Típicamente la entrada es gratis, y las que he visitado han sido amistosas y cálidas.

En este capítulo de proyecto final, nuestra meta es construir un sitio web para administrar las pláticas dadas en una reunión para compartir habilidades. Imagina un pequeño grupo de gente que se reúne regularmente en las oficinas de uno de ellos para hablar de monociclismo. El organizador previo se fue a otra ciudad, y nadie se postuló para tomar esta tarea. Queremos un sistema que deje a los participantes proponer y discutir pláticas entre ellos, sin un organizador central.

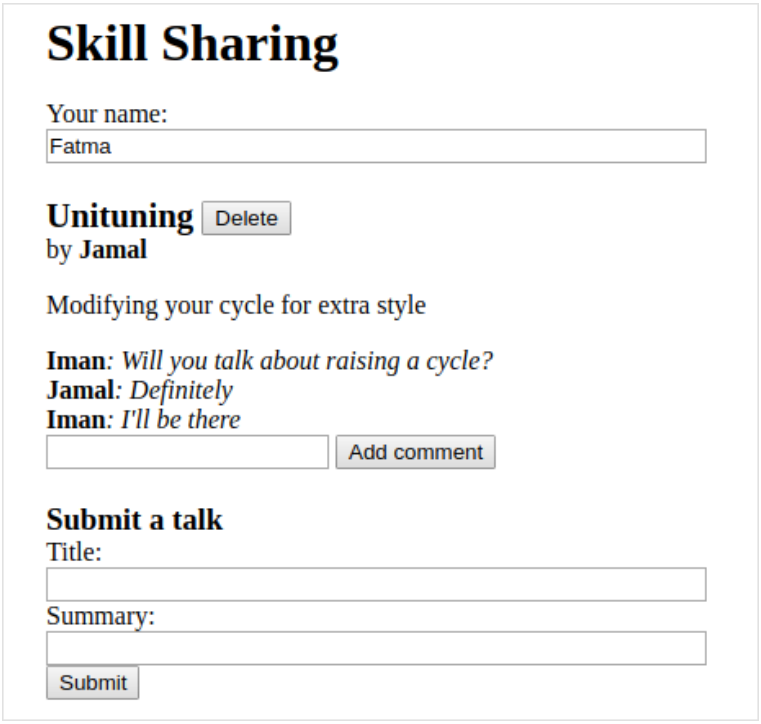
El proyecto completo puede ser descargado de *https://eloquentjavascript.net/code/skillsharing.zip* (En inglés).

Diseño

El proyecto tiene un parte de *servidor*, escrita para Node.js, y una parte *cliente*, escrita para el navegador. La parte del servidor guarda la información del sistema y se lo pasa al cliente. Además, sirve los archivos que implementan la parte cliente.

El servidor mantiene la lista de exposiciones propuestas para la próxima charla, y el cliente muestra la lista. Cada charla tiene el nombre del presentados, un título, un resumen, y una lista de comentarios asociados. el cliente permite

proponer nuevas charlas, (agregándolas a la lista), borrar charlas y comentar en las existentes. Cuando un usuario hace un cambio, el cliente hace la petición HTTP para notificar al servidor.



{{index “vista en vivo”, “experiencia de usuario”, “empujar datos”, conexión}}

La aplicación será construida para mostrar una vista *en vivo* de las charlas propuestas y sus comentarios. Cuando alguien en algún lugar manda una nueva charla o agrega un comentario, todas las personas que tienen la página abierta en sus navegadores deberían ver inmediatamente el cambio. Esto nos reta un poco: no hay forma de que el servidor abra una conexión a un cliente, ni una buena forma de saber qué clientes están viendo un sitio web.

Una solución común a este problema es llamada *sondeo largo* (le llamaremos *long polling*), que de casualidad es una de las motivaciones para el diseño de Node.

Long Polling

{{index firewall, notificación, “long polling”, red, [navegador, seguridad]}}

Para ser capaces de notificar inmediatamente a un cliente que algo ha cambiado, necesitamos una conexión con ese cliente. Como los navegadores normalmente no aceptan conexiones y los clientes están detrás de routers que blo-

quearían la conexión de todas maneras, hacer que el servidor inicie la conexión no es práctico.

Podemos hacer que el cliente abra la conexión y la mantenga de tal manera que el servidor pueda usarla para mandar información cunado lo necesite.

Pero una petición HTTP permite sólamente un flujo simple de información: el cliente manda una petición, el servidor responde con una sola respuesta, y eso es todo. Existe una tecnología moderna llamada *WebSockets*, soportada por los principales navegadores, que hace posible abrir conexiones para intercambio arbitrario de datos. Pero usarla correctamente es un poco complicado.

En este capítulo usamos una técnica más simple, el *long polling* en donde los clientes continuamente le piden al servidor nueva información usando las peticiones HTTP regulares, y el server detiene su respuesta cuando no hay nada nuevo que reportar.

Mientras el cliente se asegure de tener constantemente abierta una petición de sondeo, recibirá nueva información del servidor muy poco tiempo después de que esté disponible. Por ejemplo, si Fatma tiene nuestra aplicación abierta in su navegador, ese navegador ya habrá hecho una petición de actualizaciones y estará esperando por una respuesta a esa petición. Cuando Iman manda una charla acerca de Bajada Extrema en Monociclo, el servidor se dará cuenta de que Fatma está esperando actualizaciones y mandará una respuesta conteniendo la nueva charla, respondiendo a la petición pendiente. El navegador de Fatma recibirá los datos y actualizará la pantalla.

Para evitar que las conexiones se venzan (que sean abortadas por falta de actividad), las técnicas de *long polling* usualmente ponen el tiempo máximo para cada petición después de lo cuál el servidor responderá de todos modos aunque no tenga nada que reportar, después de lo cuál el cliente iniciará otra petición. Reiniciar periódicamente la petición hace además la técnica más robusta, permitiendo a los clientes recuperarse de fallas temporales en la conexión o problemas del servidor.

Un servidor ocupado que esté usando *long polling* podría tener miles de peticiones esperando, y por lo tanto miles de conexiones TCP abiertas. Node, que hace fácil de manejar muchas conexiones sin crear un hilo de control para cada una, es un buen elemento para nuestro sistema.

HTTP INTERFACE

Before we start designing either the server or the client, let's think about the point where they touch: the HTTP interface over which they communicate.

We will use JSON as the format of our request and response body. Like in

the file server from Chapter 17, we'll try to make good use of HTTP methods and headers. The interface is centered around the `/talks` path. Paths that do not start with `/talks` will be used for serving static files—the HTML and JavaScript code for the client-side system.

A `GET` request to `/talks` returns a JSON document like this:

```
[{"title": "Unituning",
  "presenter": "Jamal",
  "summary": "Modifying your cycle for extra style",
  "comments": []}]
```

Creating a new talk is done by making a `PUT` request to a URL like `/talks/Unituning`, where the part after the second slash is the title of the talk. The `PUT` request's body should contain a JSON object that has `presenter` and `summary` properties.

Since talk titles may contain spaces and other characters that may not appear normally in a URL, title strings must be encoded with the `encodeURIComponent` function when building up such a URL.

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

A request to create a talk about idling might look something like this:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Maureen",
 "summary": "Standing still on a unicycle"}
```

Such URLs also support `GET` requests to retrieve the JSON representation of a talk and `DELETE` requests to delete a talk.

Adding a comment to a talk is done with a `POST` request to a URL like `/talks/Unituning/comments`, with a JSON body that has `author` and `message` properties.

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{"author": "Iman",
```

```
 "message": "Will you talk about raising a cycle?"}
```

To support long polling, `GET` requests to `/talks` may include extra headers that inform the server to delay the response if no new information is available. We'll use a pair of headers normally intended to manage caching: `ETag` and `If-None-Match`.

Servers may include an `ETag` ("entity tag") header in a response. Its value is a string that identifies the current version of the resource. Clients, when they later request that resource again, may make a *conditional request* by including an `If-None-Match` header whose value holds that same string. If the resource hasn't changed, the server will respond with status code 304, which means "not modified", telling the client that its cached version is still current. When the tag does not match, the server responds as normal.

We need something like this, where the client can tell the server which version of the list of talks it has, and the server responds only when that list has changed. But instead of immediately returning a 304 response, the server should stall the response and return only when something new is available or a given amount of time has elapsed. To distinguish long polling requests from normal conditional requests, we give them another header, `Prefer: wait=90`, which tells the server that the client is willing to wait up to 90 seconds for the response.

The server will keep a version number that it updates every time the talks change and will use that as the `ETag` value. Clients can make requests like this to be notified when the talks change:

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(time passes)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295

[....]
```

The protocol described here does not do any access control. Everybody can comment, modify talks, and even delete them. (Since the Internet is full of

hooligans, putting such a system online without further protection probably wouldn't end well.)

The server

Let's start by building the server-side part of the program. The code in this section runs on Node.js.

Routing

Our server will use `createServer` to start an HTTP server. In the function that handles a new request, we must distinguish between the various kinds of requests (as determined by the method and the path) that we support. This can be done with a long chain of `if` statements, but there is a nicer way.

A *router* is a component that helps dispatch a request to the function that can handle it. You can tell the router, for example, that `PUT` requests with a path that matches the regular expression `/^\/talks\/([^\/]+)$/` (`/talks/` followed by a talk title) can be handled by a given function. In addition, it can help extract the meaningful parts of the path (in this case the talk title), wrapped in parentheses in the regular expression, and pass them to the handler function.

There are a number of good router packages on NPM, but here we'll write one ourselves to illustrate the principle.

This is `router.js`, which we will later `require` from our server module:

```
const {parse} = require("url");

module.exports = class Router {
  constructor() {
    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  resolve(context, request) {
    let path = parse(request.url).pathname;

    for (let {method, url, handler} of this.routes) {
      let match = url.exec(path);
      if (!match || request.method != method) continue;
      let urlParts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...urlParts, request);
```

```
    }
    return null;
  }
};
```

The module exports the `Router` class. A router object allows new handlers to be registered with the `add` method and can resolve requests with its `resolve` method.

The latter will return a response when a handler was found, and `null` otherwise. It tries the routes one at a time (in the order in which they were defined) until a matching one is found.

The handler functions are called with the `context` value (which will be the server instance in our case), match strings for any groups they defined in their regular expression, and the request object. The strings have to be URL-decoded since the raw URL may contain `%20`-style codes.

Serving files

When a request matches none of the request types defined in our router, the server must interpret it as a request for a file in the `public` directory. It would be possible to use the file server defined in Chapter 17 to serve such files, but we neither need nor want to support `PUT` and `DELETE` requests on files, and we would like to have advanced features such as support for caching. So let's use a solid, well-tested static file server from NPM instead.

I opted for `ecstatic`. This isn't the only such server on NPM, but it works well and fits our purposes. The `ecstatic` package exports a function that can be called with a configuration object to produce a request handler function. We use the `root` option to tell the server where it should look for files. The handler function accepts `request` and `response` parameters and can be passed directly to `createServer` to create a server that serves *only* files. We want to first check for requests that we should handle specially, though, so we wrap it in another function.

```
const {createServer} = require("http");
const Router = require("./router");
const ecstatic = require("ecstatic");

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

class SkillShareServer {
```

```
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];

    let fileServer = ecstatic({root: "./public"});
    this.server = createServer((request, response) => {
      let resolved = router.resolve(this, request);
      if (resolved) {
        resolved.catch(error => {
          if (error.status != null) return error;
          return {body: String(error), status: 500};
        }).then(({body,
                  status = 200,
                  headers = defaultHeaders}) => {
          response.writeHead(status, headers);
          response.end(body);
        });
      } else {
        fileServer(request, response);
      }
    });
  }
  start(port) {
    this.server.listen(port);
  }
  stop() {
    this.server.close();
  }
}
```

This uses a similar convention as the file server from the previous chapter for responses—handlers return promises that resolve to objects describing the response. It wraps the server in an object that also holds its state.

Talks as resources

The talks that have been proposed are stored in the `talks` property of the server, an object whose property names are the talk titles. These will be exposed as HTTP resources under `/talks/[title]`, so we need to add handlers to our router that implement the various methods that clients can use to work with them.

The handler for requests that `GET` a single talk must look up the talk and

respond either with the talk's JSON data or with a 404 error response.

```
const talkPath = /^\/talks\/([^\/]+)$/;

router.add("GET", talkPath, async (server, title) => {
  if (title in server.talks) {
    return {body: JSON.stringify(server.talks[title]),
            headers: {"Content-Type": "application/json"}};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

Deleting a talk is done by removing it from the `talks` object.

```
router.add("DELETE", talkPath, async (server, title) => {
  if (title in server.talks) {
    delete server.talks[title];
    server.updated();
  }
  return {status: 204};
});
```

The `updated` method, which we will define later, notifies waiting long polling requests about the change.

To retrieve the content of a request body, we define a function called `readStream`, which reads all content from a readable stream and returns a promise that resolves to a string.

```
function readStream(stream) {
  return new Promise((resolve, reject) => {
    let data = "";
    stream.on("error", reject);
    stream.on("data", chunk => data += chunk.toString());
    stream.on("end", () => resolve(data));
  });
}
```

One handler that needs to read request bodies is the `PUT` handler, which is used to create new talks. It has to check whether the data it was given has `presenter` and `summary` properties, which are strings. Any data coming from outside the system might be nonsense, and we don't want to corrupt our

internal data model or crash when bad requests come in.

If the data looks valid, the handler stores an object that represents the new talk in the `talks` object, possibly overwriting an existing talk with this title, and again calls `updated`.

```
router.add("PUT", talkPath,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let talk;
  try { talk = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!talk ||
      typeof talk.presenter != "string" ||
      typeof talk.summary != "string") {
    return {status: 400, body: "Bad talk data"};
  }
  server.talks[title] = {title,
                         presenter: talk.presenter,
                         summary: talk.summary,
                         comments: []};
  server.updated();
  return {status: 204};
});
```

Adding a comment to a talk works similarly. We use `readStream` to get the content of the request, validate the resulting data, and store it as a comment when it looks valid.

```
router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let comment;
  try { comment = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!comment ||
      typeof comment.author != "string" ||
      typeof comment.message != "string") {
    return {status: 400, body: "Bad comment data"};
  } else if (title in server.talks) {
    server.talks[title].comments.push(comment);
    server.updated();
    return {status: 204};
```

```
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

Trying to add a comment to a nonexistent talk returns a 404 error.

Long polling support

The most interesting aspect of the server is the part that handles long polling. When a `GET` request comes in for `/talks`, it may be either a regular request or a long polling request.

There will be multiple places in which we have to send an array of talks to the client, so we first define a helper method that builds up such an array and includes an `ETag` header in the response.

```
SkillShareServer.prototype.talkResponse = function() {
  let talks = [];
  for (let title of Object.keys(this.talks)) {
    talks.push(this.talks[title]);
  }
  return {
    body: JSON.stringify(talks),
    headers: {"Content-Type": "application/json",
              "ETag": `"${this.version}"`}
  };
};
```

The handler itself needs to look at the request headers to see whether `If-None-Match` and `Prefer` headers are present. Node stores headers, whose names are specified to be case insensitive, under their lowercase names.

```
router.add("GET", /^\/talks$/, async (server, request) => {
  let tag = /"(.*)"/.exec(request.headers["if-none-match"]);
  let wait = /\bwait=(\d+)/.exec(request.headers["prefer"]);
  if (!tag || tag[1] != server.version) {
    return server.talkResponse();
  } else if (!wait) {
    return {status: 304};
  } else {
    return server.waitForChanges(Number(wait[1]));
  }
});
```

If no tag was given or a tag was given that doesn't match the server's current version, the handler responds with the list of talks. If the request is conditional and the talks did not change, we consult the `Prefer` header to see whether we should delay the response or respond right away.

Callback functions for delayed requests are stored in the server's `waiting` array so that they can be notified when something happens. The `waitForChanges` method also immediately sets a timer to respond with a 304 status when the request has waited long enough.

```
SkillShareServer.prototype.waitForChanges = function(time) {
  return new Promise(resolve => {
    this.waiting.push(resolve);
    setTimeout(() => {
      if (!this.waiting.includes(resolve)) return;
      this.waiting = this.waiting.filter(r => r != resolve);
      resolve({status: 304});
    }, time * 1000);
  });
};
```

Registering a change with `updated` increases the `version` property and wakes up all waiting requests.

```
SkillShareServer.prototype.updated = function() {
  this.version++;
  let response = this.talkResponse();
  this.waiting.forEach(resolve => resolve(response));
  this.waiting = [];
};
```

That concludes the server code. If we create an instance of `SkillShareServer` and start it on port 8000, the resulting HTTP server serves files from the `public` subdirectory alongside a talk-managing interface under the `/talks` URL.

```
new SkillShareServer(Object.create(null)).start(8000);
```

The client

The client-side part of the skill-sharing website consists of three files: a tiny HTML page, a style sheet, and a JavaScript file.

HTML

It is a widely used convention for web servers to try to serve a file named `index.html` when a request is made directly to a path that corresponds to a directory. The file server module we use, `ecstatic`, supports this convention. When a request is made to the path `/`, the server looks for the file `./public/index.html` (`./public` being the root we gave it) and returns that file if found.

Thus, if we want a page to show up when a browser is pointed at our server, we should put it in `public/index.html`. This is our index file:

```
<!doctype html>
<meta charset="utf-8">
<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Skill Sharing</h1>

<script src="skillsharing_client.js"></script>
```

It defines the document title and includes a style sheet, which defines a few styles to, among other things, make sure there is some space between talks.

At the bottom, it adds a heading at the top of the page and loads the script that contains the client-side application.

Actions

The application state consists of the list of talks and the name of the user, and we'll store it in a `{talks, user}` object. We don't allow the user interface to directly manipulate the state or send off HTTP requests. Rather, it may emit *actions* that describe what the user is trying to do.

The `handleAction` function takes such an action and makes it happen. Because our state updates are so simple, state changes are handled in the same function.

```
function handleAction(state, action) {
  if (action.type == "setUser") {
    localStorage.setItem("userName", action.user);
```

```
    return Object.assign({}, state, {user: action.user});
  } else if (action.type == "setTalks") {
    return Object.assign({}, state, {talks: action.talks});
  } else if (action.type == "newTalk") {
    fetchOK(talkURL(action.title), {
      method: "PUT",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        presenter: state.user,
        summary: action.summary
      })
    }).catch(reportError);
  } else if (action.type == "deleteTalk") {
    fetchOK(talkURL(action.talk), {method: "DELETE"})
      .catch(reportError);
  } else if (action.type == "newComment") {
    fetchOK(talkURL(action.talk) + "/comments", {
      method: "POST",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        author: state.user,
        message: action.message
      })
    }).catch(reportError);
  }
  return state;
}
```

We'll store the user's name in `localStorage` so that it can be restored when the page is loaded.

The actions that need to involve the server make network requests, using `fetch`, to the HTTP interface described earlier. We use a wrapper function, `fetchOK`, which makes sure the returned promise is rejected when the server returns an error code.

```
function fetchOK(url, options) {
  return fetch(url, options).then(response => {
    if (response.status < 400) return response;
    else throw new Error(response.statusText);
  });
}
```

This helper function is used to build up a URL for a talk with a given title.

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

When the request fails, we don't want to have our page just sit there, doing nothing without explanation. So we define a function called `reportError`, which at least shows the user a dialog that tells them something went wrong.

```
function reportError(error) {
  alert(String(error));
}
```

Rendering components

We'll use an approach similar to the one we saw in Chapter ?, splitting the application into components. But since some of the components either never need to update or are always fully redrawn when updated, we'll define those not as classes but as functions that directly return a DOM node. For example, here is a component that shows the field where the user can enter their name:

```
function renderUserField(name, dispatch) {
  return elt("label", {}, "Your name: ", elt("input", {
    type: "text",
    value: name,
    onchange(event) {
      dispatch({type: "setUser", user: event.target.value});
    }
  }));
}
```

The `elt` function used to construct DOM elements is the one we used in Chapter ?.

A similar function is used to render talks, which include a list of comments and a form for adding a new comment.

```
function renderTalk(talk, dispatch) {
  return elt(
    "section", {className: "talk"},
    elt("h2", null, talk.title, " ", elt("button", {
      type: "button",
      onclick() {
```

```
        dispatch({type: "deleteTalk", talk: talk.title});
      }
    }, "Delete")),
    elt("div", null, "by ",
        elt("strong", null, talk.presenter)),
    elt("p", null, talk.summary),
    ...talk.comments.map(renderComment),
    elt("form", {
      onsubmit(event) {
        event.preventDefault();
        let form = event.target;
        dispatch({type: "newComment",
                  talk: talk.title,
                  message: form.elements.comment.value});
        form.reset();
      }
    }, elt("input", {type: "text", name: "comment"}), " ",
       elt("button", {type: "submit"}, "Add comment")));
}
```

The `"submit"` event handler calls `form.reset` to clear the form's content after creating a `"newComment"` action.

When creating moderately complex pieces of DOM, this style of programming starts to look rather messy. There's a widely used (non-standard) JavaScript extension called *JSX* that lets you write HTML directly in your scripts, which can make such code prettier (depending on what you consider pretty). Before you can actually run such code, you have to run a program on your script to convert the pseudo-HTML into JavaScript function calls much like the ones we use here.

Comments are simpler to render.

```
function renderComment(comment) {
  return elt("p", {className: "comment"},
             elt("strong", null, comment.author),
             ": ", comment.message);
}
```

Finally, the form that the user can use to create a new talk is rendered like this:

```
function renderTalkForm(dispatch) {
  let title = elt("input", {type: "text"});
  let summary = elt("input", {type: "text"});
```

```
  return elt("form", {
    onsubmit(event) {
      event.preventDefault();
      dispatch({type: "newTalk",
                title: title.value,
                summary: summary.value});
      event.target.reset();
    }
  }, elt("h3", null, "Submit a Talk"),
     elt("label", null, "Title: ", title),
     elt("label", null, "Summary: ", summary),
     elt("button", {type: "submit"}, "Submit"));
}
```

Polling

To start the app we need the current list of talks. Since the initial load is closely related to the long polling process—the `ETag` from the load must be used when polling—we'll write a function that keeps polling the server for `/talks` and calls a callback function when a new set of talks is available.

```
async function pollTalks(update) {
  let tag = undefined;
  for (;;) {
    let response;
    try {
      response = await fetchOK("/talks", {
        headers: tag && {"If-None-Match": tag,
                         "Prefer": "wait=90"}
      });
    } catch (e) {
      console.log("Request failed: " + e);
      await new Promise(resolve => setTimeout(resolve, 500));
      continue;
    }
    if (response.status == 304) continue;
    tag = response.headers.get("ETag");
    update(await response.json());
  }
}
```

This is an `async` function so that looping and waiting for the request is easier.

It runs an infinite loop that, on each iteration, retrieves the list of talks—either normally or, if this isn't the first request, with the headers included that make it a long polling request.

When a request fails, the function waits a moment and then tries again. This way, if your network connection goes away for a while and then comes back, the application can recover and continue updating. The promise resolved via `setTimeout` is a way to force the `async` function to wait.

When the server gives back a 304 response, that means a long polling request timed out, so the function should just immediately start the next request. If the response is a normal 200 response, its body is read as JSON and passed to the callback, and its `ETag` header value is stored for the next iteration.

The application

The following component ties the whole user interface together:

```
class SkillShareApp {
  constructor(state, dispatch) {
    this.dispatch = dispatch;
    this.talkDOM = elt("div", {className: "talks"});
    this.dom = elt("div", null,
                   renderUserField(state.user, dispatch),
                   this.talkDOM,
                   renderTalkForm(dispatch));
    this.syncState(state);
  }

  syncState(state) {
    if (state.talks != this.talks) {
      this.talkDOM.textContent = "";
      for (let talk of state.talks) {
        this.talkDOM.appendChild(
          renderTalk(talk, this.dispatch));
      }
      this.talks = state.talks;
    }
  }
}
```

When the talks change, this component redraws all of them. This is simple but also wasteful. We'll get back to that in the exercises.

We can start the application like this:

```
function runApp() {
  let user = localStorage.getItem("userName") || "Anon";
  let state, app;
  function dispatch(action) {
    state = handleAction(state, action);
    app.syncState(state);
  }

  pollTalks(talks => {
    if (!app) {
      state = {user, talks};
      app = new SkillShareApp(state, dispatch);
      document.body.appendChild(app.dom);
    } else {
      dispatch({type: "setTalks", talks});
    }
  }).catch(reportError);
}

runApp();
```

If you run the server and open two browser windows for *http://localhost:8000* next to each other, you can see that the actions you perform in one window are immediately visible in the other.

Exercises

The following exercises will involve modifying the system defined in this chapter. To work on them, make sure you download the code first (*https://eloquentjavascript.net/code/skillsharing.zip*), have Node installed *https://nodejs.org*, and have installed the project's dependency with `npm install`.

Disk persistence

The skill-sharing server keeps its data purely in memory. This means that when it crashes or is restarted for any reason, all talks and comments are lost.

Extend the server so that it stores the talk data to disk and automatically reloads the data when it is restarted. Do not worry about efficiency—do the simplest thing that works.

Comment field resets

The wholesale redrawing of talks works pretty well because you usually can't tell the difference between a DOM node and its identical replacement. But there are exceptions. If you start typing something in the comment field for a talk in one browser window and then, in another, add a comment to that talk, the field in the first window will be redrawn, removing both its content and its focus.

In a heated discussion, where multiple people are adding comments at the same time, this would be annoying. Can you come up with a way to solve it?

Exercise Hints

The hints below might help when you are stuck with one of the exercises in this book. They don't give away the entire solution, but rather try to help you find it yourself.

Estructura de Programa

Ciclo de un triángulo

Puedes comenzar con un programa que imprima los números del 1 al 7, al que puedes derivar haciendo algunas modificaciones al ejemplo de impresión de números pares dado anteriormente en el capítulo, donde se introdujo el ciclo `for`.

Ahora considera la equivalencia entre números y strings de caracteres de numeral. Puedes ir de 1 a 2 agregando 1 (`+= 1`). Puedes ir de `"#"` a `"##"` agregando un caracter (`+= "#"`). Por lo tanto, tu solución puede seguir de cerca el programa de impresión de números.

FizzBuzz

Ir a traves de los números es claramente el trabajo de un ciclo y seleccionar qué imprimir es una cuestión de ejecución condicional. Recuerda el truco de usar el operador restante (`%`) para verificar si un número es divisible por otro número (tiene un residuo de cero).

En la primera versión, hay tres resultados posibles para cada número, por lo que tendrás que crear una cadena `if`/`else if`/`else`.

La segunda versión del programa tiene una solución directa y una inteligente. La manera simple es agregar otra "rama" condicional para probar precisamente la condición dada. Para el método inteligente, crea un string que contenga la palabra o palabras a imprimir e imprimir ya sea esta palabra o el número si no hay una palabra, posiblemente haciendo un buen uso del operador `||`.

Tablero de ajedrez

El string se puede construir comenzando con un string vacío (`""`) y repetidamente agregando caracteres. Un carácter de nueva línea se escribe `"\n"`.

Para trabajar con dos dimensiones, necesitarás un ciclo dentro de un ciclo. Coloca llaves alrededor de los cuerpos de ambos ciclos para hacer fácil de ver dónde comienzan y terminan. Intenta indentar adecuadamente estos cuerpos. El orden de los ciclos debe seguir el orden en el que construimos el string (línea por línea, izquierda a derecha, arriba a abajo). Entonces el ciclo externo maneja las líneas y el ciclo interno maneja los caracteres en una sola linea.

Necesitará dos vinculaciones para seguir tu progreso. Para saber si debes poner un espacio o un signo de numeral en una posición determinada, podrías probar si la suma de los dos contadores es par (`% 2`).

Terminar una línea al agregar un carácter de nueva línea debe suceder después de que la línea ha sido creada, entonces haz esto después del ciclo interno pero dentro del bucle externo.

Funciones

Mínimo

Si tienes problemas para poner llaves y paréntesis en los lugares correctos para obtener una definición válida de función, comienza copiando uno de los ejemplos en este capítulo y modificándolo.

Una función puede contener múltiples declaraciones de `return`.

Recursión

Es probable que tu función se vea algo similar a la función interna `encontrar` en la función recursiva `encontrarSolucion` de ejemplo en este capítulo, con una cadena `if/else if/else` que prueba cuál de los tres casos aplica. El `else` final, correspondiente al tercer caso, hace la llamada recursiva. Cada una de las ramas debe contener una declaración de `return` u organizarse de alguna otra manera para que un valor específico sea retornado.

Cuando se le dé un número negativo, la función volverá a repetirse una y otra vez, pasándose a si misma un número cada vez más negativo, quedando así más y más lejos de devolver un resultado. Eventualmente quedandose sin espacio en la pila y abortando el programa.

Conteo de frijoles

TU función necesitará de un ciclo que examine cada carácter en el string. Puede correr desde un índice de cero a uno por debajo de su longitud (`< string.length`). Si el carácter en la posición actual es el mismo al que se está buscando en la función, agrega 1 a una variable contador. Una vez que el ciclo haya terminado, puedes retornat el contador.

Ten cuidado de hacer que todos las vinculaciones utilizadas en la función sean *locales* a la función usando la palabra clave `let` o `const`.

Estructuras de Datos: Objetos y Arrays

La suma de un rango

Construir un array se realiza más fácilmente al inicializar primero una vinculación a `[]` (un array nuevo y vacío) y llamando repetidamente a su método `push` para agregar un valor. No te olvides de retornar el array al final de la función.

Dado que el límite final es inclusivo, deberias usar el operador `<=` en lugar de `<` para verificar el final de tu ciclo.

El parámetro de paso puede ser un parámetro opcional que por defecto (usando el operador `=`) tenga el valor 1.

Hacer que `rango` entienda valores de paso negativos es probablemente mas facil de realizar al escribir dos ciclos por separado—uno para contar hacia arriba y otro para contar hacia abajo—ya que la comparación que verifica si el ciclo está terminado necesita ser `>=` en lugar de `<=` cuando se cuenta hacia abajo.

También puede que valga la pena utilizar un paso predeterminado diferente, es decir -1, cuando el final del rango sea menor que el inicio. De esa manera, `rango(5, 2)` retornaria algo significativo, en lugar de quedarse atascado en un ciclo infinito. Es posible referirse a parámetros anteriores en el valor predeterminado de un parámetro.

Revirtiendo un array

Hay dos maneras obvias de implementar `revertirArray`. La primera es simplemente pasar a traves del array de entrada de adelante hacia atrás y usar el metodo `unshift` en el nuevo array para insertar cada elemento en su inicio. La segundo es hacer un ciclo sobre el array de entrada de atrás hacia adelante y usar el método `push`. Iterar sobre un array al revés requiere de una especificación (algo incómoda) del ciclo `for`, como `(let i = array.length - 1; i >= 0; i--)`.

Revertir al array en su lugar es más difícil. Tienes que tener cuidado de no sobrescribir elementos que necesitarás luego. Usar `revertirArray` o de lo contrario, copiar toda el array (`array.slice(0)` es una buena forma de copiar un array) funciona pero estás haciendo trampa.

El truco consiste en *intercambiar* el primer y el último elemento, luego el segundo y el penúltimo, y así sucesivamente. Puedes hacer esto haciendo un ciclo basandote en la mitad de la longitud del array (use `Math.floor` para redondear—no necesitas tocar el elemento del medio en un array con un número impar de elementos) e intercambiar el elemento en la posición `i` con el de la posición `array.length - 1 - i`. Puedes usar una vinculación local para aferrarse brevemente a uno de los elementos, sobrescribirlo con su imagen espejo, y luego poner el valor de la vinculación local en el lugar donde solía estar la imagen espejo.

Una lista

Crear una lista es más fácil cuando se hace de atrás hacia adelante. Entonces `arrayAListа` podría iterar sobre el array hacia atrás (ver ejercicio anterior) y, para cada elemento, agregar un objeto a la lista. Puedes usar una vinculación local para mantener la parte de la lista que se construyó hasta el momento y usar una asignación como `lista = {valor: X, resto: lista}` para agregar un elemento.

Para correr a traves de una lista (en `listaAArray` y `posicion`), una especificación del ciclo `for` como esta se puede utilizar:

```
for (let nodo = lista; nodo; nodo = nodo.resto) {}
```

Puedes ver cómo eso funciona? En cada iteración del ciclo, `nodo` apunta a la sublista actual, y el cuerpo puede leer su propiedad `valor` para obtener el elemento actual. Al final de una iteración, `nodo` se mueve a la siguiente sublista. Cuando eso es nulo, hemos llegado al final de la lista y el ciclo termina.

La versión recursiva de `posición`, de manera similar, mirará a una parte más pequeña de la "cola" de la lista y, al mismo tiempo, contara atrás el índice hasta que llegue a cero, en cuyo punto puede retornar la propiedad `valor` del nodo que está mirando. Para obtener el elemento cero de una lista, simplemente toma la propiedad `valor` de su nodo frontal. Para obtener el elemento $N + 1$, toma el elemento N de la lista que este en la propiedad `resto` de esta lista.

Comparación profunda

Tu prueba de si estás tratando con un objeto real se verá algo así como `typeof x == "object" && x != null`. Ten cuidado de comparar propiedades solo cuando *ambos* argumentos sean objetos. En todo los otros casos, puede retornar inmediatamente el resultado de aplicar `===`.

Usa `Object.keys` para revisar las propiedades. Necesitas probar si ambos objetos tienen el mismo conjunto de nombres de propiedad y si esos propiedades tienen valores idénticos. Una forma de hacerlo es garantizar que ambos objetos tengan el mismo número de propiedades (las longitudes de las listas de propiedades son las mismas). Y luego, al hacer un ciclo sobre una de las propiedades del objeto para compararlos, siempre asegúrate primero de que el otro realmente tenga una propiedad con ese mismo nombre. Si tienen el mismo número de propiedades, y todas las propiedades en uno también existen en el otro, tienen el mismo conjunto de nombres de propiedad.

Retornar el valor correcto de la función se realiza mejor al inmediatamente retornar falso cuando se encuentre una discrepancia y retornar verdadero al final de la función.

Funciones de Orden Superior

Cada

Al igual que el operador `&&`, el método `every` puede dejar de evaluar más elementos tan pronto como haya encontrado uno que no coincida. Entonces la versión basada en un ciclo puede saltar fuera del ciclo—con `break` o `return`—tan pronto como se encuentre con un elemento para el cual la función predicado retorne falso. Si el ciclo corre hasta su final sin encontrar tal elemento, sabemos que todos los elementos coinciden y debemos retornar verdadero.

Para construir `cada` usando `some`, podemos aplicar las *leyes De Morgan*, que establecen que `a && b` es igual a `!(!a || !b)`. Esto puede ser generalizado a arrays, donde todos los elementos del array coinciden si no hay elemento en el array que no coincida.

Dirección de Escritura Dominante

Tu solución puede parecerse mucho a la primera mitad del ejemplo `codigosTexto`. De nuevo debes contar los caracteres por el criterio basado en `codigoCaracter`, y luego filtrar hacia afuera la parte del resultado que se refiere a caracteres sin interés (que no tengan codigos).

Encontrar la dirección con la mayor cantidad de caracteres se puede hacer con `reduce`. Si no está claro cómo, refiérate al ejemplo anterior en el capítulo, donde se usa `reduce` para encontrar el código con la mayoría de los caracteres.

La Vida Secreta de los Objetos

Un tipo vector

Mira de nuevo al ejemplo de la clase `Conejo` si no recuerdas muy bien como se ven las declaraciones de clases.

Agregar una propiedad getter al constructor se puede hacer al poner la palabra `get` antes del nombre del método. Para calcular la distancia desde (0, 0) a (x, y), puedes usar el teorema de Pitágoras, que dice que el cuadrado de la distancia que estamos buscando es igual al cuadrado de la coordenada x más el cuadrado de la coordenada y. Por lo tanto, $\sqrt{x^2 + y^2}$ es el número que quieres, y `Math.sqrt` es la forma en que calculas una raíz cuadrada en JavaScript.

Conjuntos

La forma más fácil de hacer esto es almacenar un array con los miembros del conjunto en una propiedad de instancia. Los métodos `includes` o `indexOf` pueden ser usados para verificar si un valor dado está en el array.

El constructor de clase puede establecer la colección de miembros como un array vacio. Cuando se llama a `añadir`, debes verificar si el valor dado esta en el conjunto y agregarlo, por ejemplo con `push`, de lo contrario.

Eliminar un elemento de un array, en `eliminar`, es menos sencillo, pero puedes usar `filter` para crear un nuevo array sin el valor. No te olvides de sobrescribir la propiedad que sostiene los miembros del conjunto con la versión recién filtrada del array.

El método `desde` puede usar un bucle `for/of` para obtener los valores de el objeto iterable y llamar a `añadir` para ponerlos en un conjunto recien creado.

Conjuntos Iterables

Probablemente valga la pena definir una nueva clase `IteradorConjunto`. Las instancias de Iterador deberian tener una propiedad que rastree la posición actual en el conjunto. Cada vez que se invoque a `next`, este comprueba si está hecho, y si no, se mueve más allá del valor actual y lo retorna.

La clase `Conjunto` recibe un método llamado por `Symbol.iterator` que, cuando se llama, retorna una nueva instancia de la clase de iterador para ese grupo.

Tomando un método prestado

Recuerda que los métodos que existen en objetos simples provienen de `Object.prototype`.

Y que puedes llamar a una función con una vinculación `this` específica al usar su método `call`.

Proyecto: Un Robot

Midiendo un robot

Tendrás que escribir una variante de la función `correrRobot` que, en lugar de registrar los eventos en la consola, retorne el número de pasos que le tomó al robot completar la tarea.

Tu función de medición puede, en un ciclo, generar nuevos estados y contar los pasos que lleva cada uno de los robots. Cuando has generado suficientes mediciones, puedes usar `console.log` para mostrar el promedio de cada robot, que es la cantidad total de pasos tomados dividido por el número de mediciones

Eficiencia del robot

La principal limitación de `robotOrientadoAMetas` es que solo considera un paquete a la vez. A menudo caminará de ida y vuelta por el pueblo porque el paquete que resulta estar mirando sucede que esta en el otro lado del mapa, incluso si hay otros mucho más cerca.

Una posible solución sería calcular rutas para todos los paquetes, y luego tomar la más corta. Se pueden obtener incluso mejores resultados, si hay múltiples rutas más cortas, al ir prefiriendo las que van a recoger un paquete en lugar de entregar un paquete.

Conjunto persistente

La forma más conveniente de representar el conjunto de valores de miembro sigue siendo un array, ya que son fáciles de copiar.

Cuando se agrega un valor al grupo, puedes crear un nuevo grupo con una copia del array original que tiene el valor agregado (por ejemplo, usando `concat`). Cuando se borra un valor, lo filtra afuera del array.

El constructor de la clase puede tomar un array como argumento, y almacenarlo como la (única) propiedad de la instancia. Este array nunca es actualizado.

Para agregar una propiedad (`vacio`) a un constructor que no sea un método, tienes que agregarlo al constructor después de la definición de la clase, como una propiedad regular.

Solo necesita una instancia `vacio` porque todos los conjuntos vacíos son iguales y las instancias de la clase no cambian. Puedes crear muchos conjuntos diferentes de ese único conjunto vacío sin afectarlo.

Bugs y Errores

Reintentar

La llamada a `multiplicacionPrimitiva` definitivamente debería suceder en un bloquear `try`. El bloque `catch` correspondiente debe volver a lanzar la excepción cuando no esta no sea una instancia de `FalloUnidadMultiplicadora` y asegurar que la llamada sea reintentada cuando lo es.

Para reintentar, puedes usar un ciclo que solo se rompa cuando la llamada tenga éxito, como en el ejemplo de `mirar` anteriormente en este capítulo—o usar recursión y esperar que no obtengas una cadena de fallas tan largas que desborde la pila (lo cual es una apuesta bastante segura).

La caja bloqueada

Este ejercicio requiere de un bloque `finally`. Tu función deberia primero desbloquear la caja y luego llamar a la función argumento desde dentro de cuerpo `try`. El bloque `finally` después de el debería bloquear la caja nuevamente.

Para asegurarte de que no bloqueemos la caja cuando no estaba ya bloqueada, comprueba su bloqueo al comienzo de la función y desbloquea y bloquea solo cuando la caja comenzó bloqueada.

Expresiones Regulares

Estilo entre comillas

La solución más obvia es solo reemplazar las citas con una palabra no personaje en al menos un lado. Algo como `/\W'|'\W/`. Pero también debes tener en cuenta el inicio y el final de la línea.

Además, debes asegurarte de que el reemplazo también incluya los caracteres que coincidieron con el patrón `\W` para que estos no sean dejados. Esto se puede hacer envolviéndolos en paréntesis e incluyendo sus grupos en la cadena de

reemplazo (`$1`,`$2`). Los grupos que no están emparejados serán reemplazados por nada.

Números otra vez

Primero, no te olvides de la barra invertida delante del punto.

Coincidir el signo opcional delante de el número, así como delante del exponente, se puede hacer con `[+\-]?` o `(\+|-|)` (más, menos o nada).

La parte más complicada del ejercicio es el problema hacer coincidir ambos `"5."` y `".5"` sin tambien coincidir coincidir con `"."`. Para esto, una buena solución es usar el operador `|` para separar los dos casos—ya sea uno o más dígitos seguidos opcionalmente por un punto y cero o más dígitos *o* un punto seguido de uno o más dígitos.

Finalmente, para hacer que la *e* pueda ser mayuscula o minuscula, agrega una opción `i` a la expresión regular o usa `[eE]`.

Módulos

Un robot modular

Aqui esta lo que habría hecho (pero, una vez más, no hay una sola forma *correcta* de diseñar un módulo dado):

El código usado para construir el camino de grafo vive en el módulo `grafo`. Ya que prefiero usar `dijkstrajs` de NPM en lugar de nuestro propio código de busqueda de rutas, haremos que este construya el tipo de datos de grafos que `dijkstajs` espera. Este módulo exporta una sola función, `construirGrafo` . Haria que `construirGrafo` acepte un array de arrays de dos elementos, en lugar de strings que contengan guiones, para hacer que el módulo sea menos dependiente del formato de entrada.

El módulo `caminos` contiene los datos en bruto del camino (el array `caminos`) y la vinculación `grafoCamino`. Este módulo depende de `./grafo` y exporta el grafo del camino.

La clase `EstadoPueblo` vive en el módulo `estado`. Depende del módulo `./caminos`, porque necesita poder verificar que un camino dado existe. También necesita `eleccionAleatoria`. Dado que eso es una función de tres líneas, podríamos simplemente ponerla en el módulo `estado` como una función auxiliar interna. Pero `robotAleatorio` también la necesita. Entonces tendriamos que duplicarla o ponerla en su propio módulo. Dado que esta función existe en NPM en el paquete `random-item`, una buena solución es hacer que ambos módulos

dependan de el. Podemos agregar la función `correrRobot` a este módulo también, ya que es pequeña y estrechamente relacionada con la gestión de estado. El módulo exporta tanto la clase `EstadoPueblo` como la función `correrRobot`.

Finalmente, los robots, junto con los valores de los que dependen, como `mailRoute`, podrían ir en un módulo `robots-ejemplo`, que depende de `./caminos` y exporta las funciones de robot. Para que sea posible que el `robotOrientadoAMetas` haga busqueda de rutas, este módulo también depende de `dijkstrajs`.

Al descargar algo de trabajo a los módulos de NPM, el código se volvió un poco mas pequeño. Cada módulo individual hace algo bastante simple, y puede ser leído por sí mismo. La división del código en módulos también sugiere a menudo otras mejoras para el diseño del programa. En este caso, parece un poco extraño que `EstadoPueblo` y los robots dependan de un grafo de caminos. Podría ser una mejor idea hacer del grafo un argumento para el constructor del estado y hacer que los robots lo lean del objeto estado—esto reduce las dependencias (lo que siempre es bueno) y hace posible ejecutar simulaciones en diferentes mapas (lo cual es aún mejor).

Es una buena idea usar módulos de NPM para cosas que podríamos haber escrito nosotros mismos? En principio, sí—para cosas no triviales como la función de busqueda de rutas es probable que cometas errores y pierdas el tiempo escribiendola tú mismo. Para pequeñas funciones como `eleccionAleatoria`, escribirla por ti mismo es lo suficiente fácil. Pero agregarlas donde las necesites tiende a desordenar tus módulos.

Sin embargo, tampoco debes subestimar el trabajo involucrado en *encontrar* un paquete apropiado de NPM. E incluso si encuentras uno, este podría no funcionar bien o faltarle alguna característica que necesitas. Ademas de eso, depender de los paquetes de NPM, significa que debes asegurarte de que están instalados, tienes que distribuirlos con tu programa, y podrías tener que actualizarlos periódicamente.

Entonces, de nuevo, esta es una solución con compromisos, y tu puedes decidir de una u otra manera dependiendo sobre cuánto te ayuden los paquetes.

Módulo de Caminos

Como este es un módulo CommonJS, debes usar `require` para importar el módulo grafo. Eso fue descrito como exportar una función `construirGrafo`, que puedes sacar de su objeto de interfaz con una declaración `const` de desestructuración.

Para exportar `grafoCamino`, agrega una propiedad al objeto `exports`. Ya que `construirGrafo` toma una estructura de datos que no empareja precisamente `caminos`, la división de los strings de los caminis debe ocurrir en tu módulo.

Dependencias circulares

El truco es que `require` agrega módulos a su caché *antes* de comenzar a cargar el módulo. De esa forma, si se realiza una llamada `require` mientras está ejecutando el intento de cargarlo, ya es conocido y la interfaz actual sera retornada, en lugar de comenzar a cargar el módulo una vez más (lo que eventualmente desbordaría la pila).

Si un módulo sobrescribe su valor `module.exports`, cualquier otro módulo que haya recibido su valor de interfaz antes de que termine de cargarse ha conseguido el objeto de interfaz predeterminado (que es probable que este vacío), en lugar del valor de interfaz previsto.

Programación Asincrónica

Siguiendo el bisturí

Esto se puede realizar con un solo ciclo que busca a través de los nidos, avanzando hacia el siguiente cuando encuentre un valor que no coincida con el nombre del nido actual, y retornando el nombre cuando esta encuentra un valor que coincida. En la función `async`, un ciclo regular `for` o `while` puede ser utilizado.

Para hacer lo mismo con una función simple, tendrás que construir tu ciclo usando una función recursiva. La manera más fácil de hacer esto es hacer que esa función retorne una promesa al llamar a `then` en la promesa que recupera el valor de almacenamiento. Dependiendo de si ese valor coincide con el nombre del nido actual, el controlador devuelve ese valor o una promesa adicional creada llamando a la función de ciclo nuevamente.

No olvides iniciar el ciclo llamando a la función recursiva una vez desde la función principal.

En la función `async`, las promesas rechazadas se convierten en excepciones por `await` Cuando una función `async` arroja una excepción, su promesa es rechazada. Entonces eso funciona.

Si implementaste la función no-`async` como se describe anteriormente, la forma en que `then` funciona también provoca automáticamente que una falla termine en la promesa devuelta. Si una solicitud falla, el manejador pasado a `then` no se llama, y la promesa que devuelve se rechaza con la misma razón.

Construyendo Promise.all

La función pasada al constructor `Promise` tendrá que llamar `then` en cada una de las promesas del array dado. Cuando una de ellas tenga éxito, dos cosas

deben suceder. El valor resultante debe ser almacenado en la posición correcta de un array de resultados, y debemos verificar si esta fue la última promesa pendiente y terminar nuestra promesa si asi fue.

Esto último se puede hacer con un contador que se inicializa con la longitud del array de entrada y del que restamos 1 cada vez que una promesa tenga éxito. Cuando llega a 0, hemos terminado. Asegúrate de tener en cuenta la situación en la que el array de entrada este vacío (y por lo tanto ninguna promesa nunca se resolverá).

El manejo de la falla requiere pensar un poco, pero resulta ser extremadamente sencillo. Solo pasa la función `reject` de la promesa de envoltura a cada una de las promesas en el array como manejador `catch` o como segundo argumento a `then` para que una falla en una de ellos desencadene el rechazo de la promesa de envoltura completa.

Proyecto: Un Lenguaje de Programación

Arrays

The easiest way to do this is to represent Egg arrays with JavaScript arrays.

The values added to the top scope must be functions. By using a rest argument (with triple-dot notation), the definition of `array` can be *very* simple.

Closure

Again, we are riding along on a JavaScript mechanism to get the equivalent feature in Egg. Special forms are passed the local scope in which they are evaluated so that they can evaluate their subforms in that scope. The function returned by `fun` has access to the `scope` argument given to its enclosing function and uses that to create the function's local scope when it is called.

This means that the prototype of the local scope will be the scope in which the function was created, which makes it possible to access bindings in that scope from the function. This is all there is to implementing closure (though to compile it in a way that is actually efficient, you'd need to do some more work).

Comments

Make sure your solution handles multiple comments in a row, with potentially whitespace between or after them.

A regular expression is probably the easiest way to solve this. Write something that matches "whitespace or a comment, zero or more times". Use the `exec` or `match` method and look at the length of the first element in the returned array (the whole match) to find out how many characters to slice off.

Fixing scope

You will have to loop through one scope at a time, using `Object.getPrototypeOf` to go the next outer scope. For each scope, use `hasOwnProperty` to find out whether the binding, indicated by the `name` property of the first argument to `set`, exists in that scope. If it does, set it to the result of evaluating the second argument to `set` and then return that value.

If the outermost scope is reached (`Object.getPrototypeOf` returns null) and we haven't found the binding yet, it doesn't exist, and an error should be thrown.

El Modelo de Objeto del Documento

Construye una tabla

Puedes utilizar `document.createElement` para crear nuevos nodos elemento, `document.createTextNode` para crear nuevos nodos de texto, y el método `appendChild` para poner nodos dentro de otros nodos.

Querrás recorrer los nombres de las llaves una vez para llenar la fila superior y luego nuevamente para cada objeto en el arreglo para construir las filas de datos. Para obtener un arreglo con los nombres de las llaves proveniente del primer objeto, la función `Object.keys` será de utilidad.

Para agregar la tabla al nodo padre correcto, puedes utilizar `document.getElementById` o `document.querySelector` para encontrar el nodo con el atributo `id` adecuado.

Elementos por nombre de tag

La solución es expresada de manera más sencilla con una función recursiva, similar a la función `hablaSobre` definida anteriormente en este capítulo.

Puedes llamar a `byTagname` recursivamente, concatenando los arreglos resultantes para producir la salida. O puedes crear una función interna que se llama a sí misma recursivamente y que tiene acceso a un arreglo definido en la función exterior, al cual se puede agregar los elementos coincidentes que encuentre. No

olvides llamar a la función interior una vez desde la función exterior para iniciar el proceso.

La función recursiva debe revisar el tipo de nodo. En este caso solamente estamos interesados por los nodos de tipo 1 (`Node.ELEMENT_NODE`). Para tales nodos, debemos de iterar sobre sus hijos y, para cada hijo, observar si los hijos coinciden con la consulta mientras también se realiza una llamada recursiva en él para inspeccionar a sus propios hijos.

El sombrero del gato

Las funciones `Math.cos` y `Math.sin` miden los ángulos en radianes, donde un círculo completo es 2π. Para un ángulo dado, puedes obtener el ángulo inverso agregando la mitad de esto, que es `Math.PI`. Esto puede ser útil para colocar el sombrero en el lado opuesto de la órbita.

Manejo de Eventos

Globo

Querrás registrar un manejador para el evento `"keydown"` y mirar `event.key` para averiguar si se presionó la teclas de flecha hacia arra o hacia abajo.

El tamaño actual se puede mantener en una vinculación para que puedas basar el nuevo tamaño en él. Será útil definir una función que actualice el tamaño, tanto el enlace como el estilo del globo en el DOM, para que pueda llamar desde su manejador de eventos, y posiblemente también una vez al comenzar, para establecer el tamaño inicial.

Puedes cambiar el globo a una explosión reemplazando el texto del nodo con otro (usando `replaceChild`) o estableciendo la propiedad `textContent` de su no padre a una nueva cadena.

Mouse trail

La creación de los elementos se realiza de mjor manera con un ciclo. Añadelos al documento para que aparezcan. Para poder acceder a ellos más tarde para cambiar su posición, querrás alamacenar los elementos en una matriz.

Se puede hacer un ciclo a través de ellos manteniendo una variable contador y agregando 1 cada vez que se activa el evento `"mousemove"`. El operador restante (`% elements.length`) pueden ser usado para obtener un índice de matriz válido para elegir el elemento que se desaea colocar durante un evento determinado.

Otro efecto interesante se puede lograr modelando un sistema simple físico. Utiliza el evento `"mousemove"` para actualizar un par de enlaces que rastrean la posición del mouse. Luego usa `requestAnimationFrame` para simular que los elementos finales son atraidos a la posición a la posición del puntero del mouse. En cada paso de la animación, actualiza la posición en función de su posición relativa al puntero (y, opcionalmente, una velocidad que se alamacena para cada elemento). Averiguar una buena forma de hacerlo depende de ti.

Pestañas

Un error con el que puedes encontrarte es que no puede usar directamente la propiedad `childNodes` del nodo como una colección de nodos de pestañas. Por un lado, cuando agregas los botones, también se convertirán en nodos secundarios y terminarán en este objeto porque es una estructura de datos en vivo. Por otro lado, los nodos de texto creados para el espacio en blanco entre los nodos también están en `childNodes` pero no deberían tener sus propias pestañas. Puedes usar `children` en lugar de`childNodes` para ignorar los nodos con texto.

Puedes comenzar creando una colección de pestañas para que tengas fácil acceso a ellas. Para implementar el estilo de los botones, puedes almacenar objetos que contengan tanto el panel de pestañas como su botón.

Yo recomiendo escribir una función aparte para cambiar las pestañas. Puedes almacenar la pestaña seleccionada anteriormente y cambiar solo los estilos necesarios para ocultarla y mostrar la nueva, o simplemente puedes actualizar el estilo de todas las pestañas cada vez que se selecciona una nueva pestaña.

Es posible que desees llamar a esta función inmediatamente para que la interfaz comience con la primera pestaña visible.

Proyecto: Un Juego de Plataforma

Pausing the game

An animation can be interrupted by returning `false` from the function given to `runAnimation`. It can be continued by calling `runAnimation` again.

So we need to communicate the fact that we are pausing the game to the function given to `runAnimation`. For that, you can use a binding that both the event handler and that function have access to.

When finding a way to unregister the handlers registered by `trackKeys`, remember that the *exact* same function value that was passed to `addEventListener` must be passed to `removeEventListener` to successfully remove a handler.

Thus, the `handler` function value created in `trackKeys` must be available to the code that unregisters the handlers.

You can add a property to the object returned by `trackKeys`, containing either that function value or a method that handles the unregistering directly.

A monster

If you want to implement a type of motion that is stateful, such as bouncing, make sure you store the necessary state in the actor object—include it as constructor argument and add it as a property.

Remember that `update` returns a *new* object, rather than changing the old one.

When handling collision, find the player in `state.actors` and compare its position to the monster's position. To get the *bottom* of the player, you have to add its vertical size to its vertical position. The creation of an updated state will resemble either `Coin`'s `collide` method (removing the actor) or `Lava` 's (changing the status to `"lost"`), depending on the player position.

Node.js

Search tool

Your first command line argument, the regular expression, can be found in `process.argv[2]`. The input files come after that. You can use the `RegExp` constructor to go from a string to a regular expression object.

Doing this synchronously, with `readFileSync`, is more straightforward, but if you use `fs.promises` again to get promise-returning functions and write an `async` function, the code looks similar.

To figure out whether something is a directory, you can again use `stat` (or `statSync`) and the stats object's `isDirectory` method.

Exploring a directory is a branching process. You can do it either by using a recursive function or by keeping an array of work (files that still need to be explored). To find the files in a directory, you can call `readdir` or `readdirSync`. The strange capitalization—Node's file system function naming is loosely based on standard Unix functions, such as `readdir`, that are all lowercase, but then it adds `Sync` with a capital letter.

To go from a filename read with `readdir` to a full path name, you have to combine it with the name of the directory, putting a slash character (`/`) between them.

Directory creation

You can use the function that implements the `DELETE` method as a blueprint for the `MKCOL` method. When no file is found, try to create a directory with `mkdir`. When a directory exists at that path, you can return a 204 response so that directory creation requests are idempotent. If a nondirectory file exists here, return an error code. Code 400 ("bad request") would be appropriate.

A public space on the web

You can create a `<textarea>` element to hold the content of the file that is being edited. A `GET` request, using `fetch`, can retrieve the current content of the file. You can use relative URLs like *index.html*, instead of *http://localhost:8000/index.html*, to refer to files on the same server as the running script.

Then, when the user clicks a button (you can use a `<form>` element and `"submit"` event), make a `PUT` request to the same URL, with the content of the `<textarea>` as request body, to save the file.

You can then add a `<select>` element that contains all the files in the server's top directory by adding `<option>` elements containing the lines returned by a `GET` request to the URL `/`. When the user selects another file (a `"change"` event on the field), the script must fetch and display that file. When saving a file, use the currently selected filename.

Proyecto: Sitio web para compartir habilidades

Disk persistence

The simplest solution I can come up with is to encode the whole `talks` object as JSON and dump it to a file with `writeFile`. There is already a method (`updated`) that is called every time the server's data changes. It can be extended to write the new data to disk.

Pick a filename, for example `./talks.json`. When the server starts, it can try to read that file with `readFile`, and if that succeeds, the server can use the file's contents as its starting data.

Beware, though. The `talks` object started as a prototype-less object so that the `in` operator could reliably be used. `JSON.parse` will return regular objects with `Object.prototype` as their prototype. If you use JSON as your file format, you'll have to copy the properties of the object returned by `JSON.parse` into a new, prototype-less object.

Comment field resets

The best way to do this is probably to make talks component objects, with a `syncState` method, so that they can be updated to show a modified version of the talk. During normal operation, the only way a talk can be changed is by adding more comments, so the `syncState` method can be relatively simple.

The difficult part is that, when a changed list of talks comes in, we have to reconcile the existing list of DOM components with the talks on the new list—deleting components whose talk was deleted and updating components whose talk changed.

To do this, it might be helpful to keep a data structure that stores the talk components under the talk titles so that you can easily figure out whether a component exists for a given talk. You can then loop over the new array of talks, and for each of them, either synchronize an existing component or create a new one. To delete components for deleted talks, you'll have to also loop over the components and check whether the corresponding talks still exist.

Index

! operator, 19, 33
!= operator, 18
!== operator, 21
* operator, 13, 20, 153
= operator, 35
+ operator, 13, 16, 20, 153
++ operator, 36
+= operator, 35, 210
− operator, 14, 17, 20
−− operator, 36
−= operator, 35
/ operator, 14
/= operator, 35
< operator, 17
<= operator, 18
= operator, 25, 49, 65, 167, 170, 220
== operator, 18, 21, 68, 85, 203
=== operator, 21, 85, 120, 333
> operator, 17
>= operator, 18
?: operator, 19, 22, 219
[] (array), 61
[] (subscript), 61, 62
[network, stack], 204
% operator, 14, 35, 329, 330, 342
&& operator, 18, 22, 100
| | operator, 19, 21, 53, 100, 329
{} (block), 30
{} (object), 64, 68

200 (HTTP status code), 298, 302
204 (HTTP status code), 305, 306
304 (HTTP status code), 313, 320, 326
400 (HTTP status code), 345
403 (HTTP status code), 303
404 (HTTP status code), 303, 316, 319
405 (HTTP status code), 302
500 (HTTP status code), 302

a (HTML tag), 236, 238
Abelson, Hal, 213
absolute positioning, 242, 246, 255, 259, 266
absolute value, 80
abstraccion, 230
abstraccione, 87
abstracción, 213
abstract data type, 101
abstract syntax tree, *see* syntax tree
abstraction, 41, 86, 87, 89
acceleration, 285
access control, 101, 149, 313
actor, 271, 277, 284
add method, 120
addEntry function, 68
addEventListener method, 249, 286, 300
addition, 13, 120

adoption, 150
advertencia, 294
ages example, 109
aislamiento, 178
alcace, 42
alcance, 43, 178, 179, 181, 182
alcance global, 178, 262
alcance léxico, 44
alphanumeric character, 152
alt attribute, 233
alt key, 254
altKey property, 254
ambiguity, 226
American English, 153
analysis, 134, 139
ancestor element, 278
Android, 254
animaciones, 268
animación, 259
animation, 243, 266, 273
  platform game, 281, 285–287, 343
  spinning cat, 242, 246
antirrebote, 263
anyStorage function, 210, 212
apio, 309
aplicación, 1, 310
appendChild method, 233, 341
application (of functions), *see* function application
aprender, 7
aprendizaje, 8, 309
archivo, 176, 183, 292
archivos, 181
argument, 48, 162
argumento, 27, 53, 77, 213
arguments object, 331
argv property, 292
arithmetic, 13, 20, 221
arrastrar, 256
array, 62–64, 66, 78, 81, 84, 88, 98, 100, 113, 128, 147, 154, 185, 225, 334, 335
  as matrix, 270
  as table, 70
  creation, 61, 331
  filtering, 91
  indexing, 61, 71, 75, 331, 342
  iteration, 71, 90
  length of, 63
  methods, 74, 83, 90–93, 96, 99, 100
  representation, 81
  searching, 71, 75
  traversal, 88
Array prototype, 104, 108
array-like object, 230, 231
Array.from function, 205, 231, 293
arrays, 96
arrays in egg (exercise), 225, 340
arroba, 269
arrow function, 46, 103
arrow key, 265
artificial intelligence, 122, 224
artificial life, 268
asignación, 167
assert function, 147
assertion, 147
assignment, 25, 35, 170, 226, 341
assumption, 147
asterisk, 13, 153
async function, 206, 207, 210, 212, 325
asynchronous programming, 189, 190, 194, 195, 206, 208, 209, 288
  in Node.js, 290, 297, 299, 304, 307
atributo, 235
attribute, 235
attributo, 230
automatic semicolon insertion, 24

automation, 132, 137
autómata, 122
avatar, 268
average function, 95
await keyword, 206, 208, 210
axis, 284

Babbage, Charles, 60
background, 276, 282
background (CSS), 266, 268, 277
backslash character, 15, 150, 152, 165, 337
  as path separator, 303
backtick, 15, 16
backtracking, 159, 160, 163
balloon (exercise), 265, 342
banking example, 143
Banks, Ian, 267
barra de progreso, 259
bean counting (exercise), 59, 331
beforeunload event, 261
behavior, 224
benchmark, 237
best practices, 3
biblioteca, 294
big ball of mud, 175
binary data, 3, 11
binary number, 11, 12, 138, 159
binary operator, 13, 17
binding, 220, 221, 224, 226
  assignment, 25, 45
  definition, 24, 226, 341
  from parameter, 42, 51
  global, 42, 135, 289, 291, 292
  local, 43
  model of, 25, 67
  naming, 26, 37, 54, 79, 136
  scope of, 42
  visibility, 43
bit, 4, 11, 12, 17
bitfield, 257
bits, 70
block, 41, 142, 144, 214
block comment, 38, 163
block element, 236, 238
blocking, 190, 243, 263, 297
bloque, 30, 43, 46, 65, 239
bloques, 33
blur event, 260
body (HTML tag), 228
body (HTTP), 298, 305, 306, 317
body property, 228, 229, 232
Book of Programming, 11, 175, 290
Boolean, 17, 29, 32, 66, 219, 221
  conversion to, 21, 29, 33
Boolean function, 29
Booleano, 151
border (CSS), 236, 239
border-radius (CSS), 255
botón del mouse, 250, 251, 255
bouncing, 269, 273, 281, 284
boundary, 157, 169, 173, 336
box shadow (CSS), 278
braces, *see* curly braces
branching, 158, 160
branching recursion, 52
break keyword, 35, 37
British English, 153
broadcastConnections function, 202
browser, 6, 249
  window, 249
bubbling, *see* event propagation
Buffer class, 296, 299, 300
bug, 134, 166, 173, 176
building Promise.all (exercise), 212, 339
button, 248
button (HTML tag), 249, 254, 266
button property, 250, 257
buttons property, 257

cache, 181
caché, 193
caja, 149, 227, 268
caja de arena, 60, 227
call method, 103, 108
call stack, 47, 50, 53, 141, 142, 145
callback function, 190, 196, 198, 200, 248, 286, 287, 296, 297, 299, 320, 325
calling (of functions), *see* function application
camel case, 37, 240
campo, 254
campo de texto, 260
cancelAnimationFrame function, 263
canvas, 268
capas, 204
capitalization, 37, 106, 154, 240, 246, 300
capture group, 155, 157, 162, 315
capítulo de proyecto, 309
caracteres chinos, 97
caracteres de nueva línea, 270
caret character, 152, 157, 169, 295
carrera, 267
carriage return, 168
carácter, 96
carácter de tubería, 158
carácteres de punto, 28
carácteres de tabulación, 33
Cascading Style Sheets, *see* CSS
case conversion, 63
case keyword, 37
case sensitivity, 154, 337
casual computing, 1
cat's hat (exercise), 246
catch keyword, 141, 142, 145, 146, 148, 208, 336
catch method, 197
CD, 11
Celsius, 116
center, 280
centering, 243
change event, 345
character, 15, 16, 97
character category, 171
character encoding, 296
characterCount function, 94
characterScript function, 99, 100, 333
charCodeAt method, 97
chess board (exercise), 40, 330
chicks function, 210
child node, 231, 232
childNodes property, 231, 234, 343
children property, 231
Chinese characters, 98
ciclo, 31, 39, 88, 95, 199, 228, 330, 331
ciclo infinito, 35, 331
cierre, 50
circo, 74
circular dependency, 187, 339
clase, 105, 106, 120, 124, 269
class attribute, 232, 236, 240, 275, 277, 278
class declaration
  properties, 107
class hierarchy, 118
className property, 236
cleaning up, 143
clearInterval function, 263
clearTimeout function, 262, 263
click event, 249, 251, 255, 257
client, 299, 321
cliente, 309
clientHeight property, 236
clientWidth property, 236
clientX property, 255, 258
clientY property, 255, 258
closure, 225, 340, 341, 343

closure in egg (exercise), 225, 340
code
  structure of, 23
code golf, 173
code structure, 33, 41, 175, 184
codePointAt method, 97
codiciosos, 164
codificación de caracteres, 296
coercion de tipo, 20
coin, 268, 284
Coin class, 273, 284
colección, 6, 61
collection, 63, 66, 84
collision detection, 281, 284, 285, 344
colon character, 19, 36, 64, 239
color (CSS), 239
comentario, 38, 163, 229, 309
comillas, 174
comma character, 213
command key, 254
command line, 177, 290, 292, 307
comment, 81, 168, 226, 312, 318, 323, 340
comment field reset (exercise), 328, 346
COMMENT_NODE code, 229
comments in egg (exercise), 226, 340
CommonJS, 292, 293
CommonJS modules, 179, 180
compareRobots function, 132
comparison, 17, 21, 32, 37, 85, 221, 331
  of NaN, 18
  of numbers, 17, 28
  of objects, 68
  of strings, 18
  of undefined values, 21
compartir habilidades, 309
compatibilidad, 7
compatibility, 295
compilation, 183, 223, 224, 340
complejidad, 3, 86
complexity, 4, 118, 160, 241, 275
comportamiento, 173
composability, 6, 95, 184
computadora, 1, 3
computed property, 62
concat method, 75, 100, 335, 341
concatenation, 16, 75, 341
conditional execution, 19, 36, 40, 219
conditional operator, 19, 22, 219
conditional request, 313
conexiones, 311
conexión, 310, 311
configuración, 168
conjunto, 151, 229
conjunto de datos, 71, 90, 91
connected graph, 131
connections binding, 202
consistencia, 37
consistency, 230
consola de JavaScript, 28, 291
console.log, 6, 10, 17, 28, 47, 49, 57, 139, 291, 301
const keyword, 26, 43, 67, 79, 81
constant, 285
constante, 26
constantes, 79
constructor, 37, 106, 118, 133, 142, 156, 164, 334, 335
constructora, 105, 135
constructoras, 106
Content-Type header, 298, 302, 303, 308
continuación, 192
continue keyword, 35
control flow, 29, 31, 33, 34, 47, 141, 190, 206
control key, 254
convención, 37

coordenadas, 255
coordinates, 120, 244, 276, 280, 282
copy-paste programming, 55, 176
copyright, 177
corchete, 112
corchetes, 61, 81, 152
corredores de pruebas, 138
correlaciones, 73
correlación, 73
correlaciónes, 71
correlation, 69, 70
cosine, 79, 244
countBy function, 98, 100
counter variable, 32, 34, 244, 330, 331, 340
CPU, 190
crash, 145, 147, 318, 327
createElement method, 234, 341
createReadStream function, 300, 304
createServer function, 297, 298, 300, 314, 315
createTextNode method, 233
createWriteStream function, 300, 306
crow-tech module, 193
crying, 154
CSS, 239–241, 275, 277–280, 321
ctrlKey property, 254
cuadrado, 29
cuadro de diálogo, 27
cuadrícula, 268
cuervo, 191, 193, 205
curl program, 306
curly braces, 5, 41, 46, 64, 68, 81, 89, 153
cutting point, 275
cwd function, 303
código, 8, 163, 267
córvidos, 191

Dark Blue (game), 267
dash character, 14, 152
data, 2, 11
data attribute, 235, 266
data event, 300
data format, 81
data loss, 327
data structure, 60, 61, 84, 109, 126
  tree, 228
Date class, 156, 178, 179
date-names package, 179
Date.now function, 156
datos, 60
datos binarios, 296
dblclick event, 255
debugger statement, 139
debugging, 7, 134, 136, 138, 139, 142, 146, 147, 173
decimal number, 11, 138, 159
declaración, 24, 29, 32, 42, 65
declaración de clase, 107
declaraciónes, 34
decodeURIComponent function, 302, 315
deep comparison, 68, 85
deep comparison (exercise), 85, 333
default behavior, 238, 252
default export, 182
default keyword, 37
default value, 22, 49
defineProperty function, 334
definirTipoSolicitud function, 194, 199
DELETE method, 301, 304, 317
delete method, 120
delete operator, 65
dependencia, 69
dependency, 175, 176, 178, 182, 187, 294, 295
depuración, 135
desbordar, 12
descarga, 309

descargar, 293
desenrollando la pila, 141
deserialization, 82
design, 176
destructuring, 157
destructuring binding, 80, 180, 338
developer tools, 8, 28, 145
diagrama de flujo, 158
dialecto, 183
diario, 68
Dijkstra's algorithm, 185
Dijkstra, Edsger, 122, 185
dijkstrajs package, 185, 337
dimensiones, 330
dimensions, 120, 236, 267, 268, 281
dinosaur, 224
dirección, 81
direct child node, 241
direction (writing), 100
directorio, 292, 295, 297
directory, 292, 296, 301, 303, 304, 307, 345
directory creation (exercise), 307, 345
disco duro, 189
discretization, 268, 281, 287
dispatch, 314
dispatching, 36
display, 275, 287, 288
display (CSS), 239, 266
division, 14
division by zero, 14
do loop, 33, 128
Document Object Model, *see* DOM
documentElement property, 228
documento, 227, 261
dollar sign, 26, 157, 162, 169
DOM, 228, 235
  attributes, 235
  construction, 230, 233, 234
  events, 249, 254
  graphics, 268, 275, 277, 278
  interface, 229
  modification, 233
  querying, 232, 241
  tree, 228
domain-specific language, 86, 138, 150, 225, 241
DOMDisplay class, 275, 276
dominant direction (exercise), 100, 333
dot character, *see* period character
double click, 255
double-quote character, 15, 174, 213
download, 8, 177, 306, 327
draggable bar example, 256
drawing, 227, 236, 237, 243, 275
drawing program example, 255
duplication, 176
dígito, 138, 151–153, 155
dígitos, 154

ECMAScript, 7, 182
ECMAScript 6, 7
ecstatic package, 315
editores, 34
efecto secundario, 24, 28, 42, 233
efficiency, 52, 83, 95, 202, 223, 237, 278
eficiencia, 268
Egg language, 213, 217–219, 221, 222, 224–226, 229
ejecución condicional, 29
ejemplo de la granja, 57
ejercicios, 2, 8, 138
elección, 158
electronic life, 268
elegance, 215
elegancia, 52
element, 229, 231
ELEMENT_NODE code, 229, 342

elemento, 229, 234
elipse, 243
ellipse, 244
else keyword, 30
elt function, 234, 323
emoji, 16, 97, 171, 265
empaquetadores, 184
empty set, 163
encapsulación, 102, 118
encapsulation, 101, 110, 249, 275
encodeURIComponent function, 312, 322
end event, 300
end method, 298, 299, 302
enemies example, 168
enlace, 230, 232, 252, 254
ENOENT (status code), 304
entorno, 27
entrada, 268
enum (reserved word), 26
environment, 219
equality, 18
error, 97, 134, 135, 138, 140, 141, 145, 146, 196, 198, 204
error de sintaxis, 26
error event, 306
error handling, 134, 141, 145, 297, 302, 304, 323, 326
error message, 218
error recovery, 140
error response, 302, 305
Error type, 142, 145, 146, 304
errores, 164
ES modules, 182
escape key, 289
escaping
  in regexps, 150, 152, 165
  in strings, 15, 213
  in URLs, 302, 312, 315
escribiendo, 263
espacio en blanco, 213
espacios en blanco, 216
especificidad, 241
espías, 259
estado, 24, 124, 202, 209
estructura, 227
estructura de datos, 64, 186, 214, 227
estándar, 7, 27, 37, 92, 170, 292
ETag header, 313, 319, 326
etiqueta, 227, 240
eval, 178
evaluación de corto circuito, 22
evaluate function, 218, 219, 221
evaluation, 178, 218, 224
even number, 58
event handling, 248–253, 259–261, 268, 286, 288, 289, 300, 343
event loop, 208
event object, 255
event propagation, 251, 260, 261
event type, 250
every method, 100
everything (exercise), 100, 333
everywhere function, 201
evolución convergente, 192
evolution, 150, 295
exception handling, 142, 143, 145–149, 196, 197, 206, 208, 212, 339
exception safety, 144
exec method, 154, 155, 166, 167
execution order, 29, 45, 47
exercises, 39
exit method, 291
expectation, 252
experiment, 3
experimentar, 8, 173
exponent, 13, 174, 337
exponente, 337

exponentiation, 32, 34
export keyword, 182
exports object, 179, 181, 182, 293, 338, 339
exposiciones, 309
expresion, 45
expresiones regular, 216
expresiones regulare, 150
expresiones regulares, 164, 168
expresión, 23, 24, 28, 32, 34, 213, 214
expresión regular, 151, 173
expression, 218
expressivity, 225
extension, 292
extiende, 78
extraction, 155

factorial function, 9
Fahrenheit, 116
fallthrough, 37
false, 17
farm example, 55, 158
fecha, 152, 154–156
fetch function, 299, 322, 325, 345
field, 328
fila, 246
file, 304, 345
  access, 296
  resource, 301, 303
  stream, 300
file extension, 303
file format, 168
file server, 321
file server example, 301, 303–308, 345
file size, 184
file system, 296, 301, 303, 345
filter method, 91, 92, 95, 99, 125, 201, 333–335
finally keyword, 144, 149, 336
findIndex method, 98
findInStorage function, 205, 206
findRoute function, 130, 203
finish event, 306
firstChild property, 231
fixed positioning, 259
fixing scope (exercise), 226, 341
FizzBuzz (exercise), 40, 329
flattening (exercise), 100
flexibility, 7
floating-point number, 13
flooding, 202
flow diagram, 159
flujo de control, 90, 143
foco, 254, 260
focus, 328
focus event, 260
fold, *see* reduce method
fondo, 268
font-family (CSS), 240
font-size (CSS), 265
font-weight (CSS), 240
for loop, 34, 35, 71, 88, 100, 146, 331, 332
for/of loop, 72, 97, 111, 113, 115, 334
forEach method, 90
form, 308
form (HTML tag), 324, 345
formatDate module, 179, 182
formato de datos, 230
fractional number, 13, 174, 268
framework, 57
fs package, 296, 297
funcion, 213
funciones de flecha, 210
función, 27, 214
función de devolución de llamada, 192, 194
función de predicado, 99

función interior, 342
function, 27, 41, 46, 135, 222
  application, 27, 28, 42, 47, 48, 51, 52, 92, 145, 213, 219
  as property, 63
  as value, 41, 45, 50, 88, 89, 92, 250, 286, 343
  body, 41, 46
  callback, *see* callback function
  declaration, 45
  definition, 41, 45, 54
  higher-order, 45, 88, 89, 91–93, 95, 162, 286
  model of, 51
  naming, 54, 56
  purity, 57
  scope, 44, 178, 225
function application, 78
Function constructor, 179, 181, 221, 224
function keyword, 41, 45
Function prototype, 104, 108
futuras, 26
future, 7, 45
físico, 343

game, 286, 288, 289
  screenshot, 280
GAME_LEVELS data set, 288
garbage collection, 12
garble example, 293
generador, 207
GET method, 299, 301, 303, 312, 316
get method, 110
getAttribute method, 235, 236
getBoundingClientRect method, 237
getDate function, 157
getDate method, 156
getElementById method, 232, 341
getElementsByClassName method, 232
getElementsByTagName method, 232, 234, 246, 341
getFullYear method, 156
getHours method, 156
getMinutes method, 156
getMonth method, 156
getPrototypeOf function, 104, 106, 226, 341
getSeconds method, 156
getter, 115, 120, 273
getTime method, 156
getYear method, 156
global object, 135
global scope, 42, 221, 291, 292, 341
globo, 265
goalOrientedRobot function, 131
gossip property, 201
grafo, 123, 130, 185, 203
grammar, 23, 168
gramática, 134
gran bola de barro, 175
graphics, 268, 275, 278
grave accent, *see* backtick
gravity, 285
greater than, 17
greed, 163
grep, 307
grid, 276, 282
Group class, 120, 132, 207, 334
groupBy function, 100
grouping, 14, 30, 154, 155, 162, 336
groups (exercise), 120, 334
grupo de usuarios, 309
gráficos, 268

h1 (HTML tag), 236
handleAction function, 321
hard disk, 184, 192
hard drive, 11, 327

hard-coding, 232
has method, 110, 120
hash character, 226
hasOwnProperty method, 110, 226, 341
head (HTML tag), 228
head property, 228
header, 298, 312
help text example, 260
herencia, 117, 118
herramienta, 150
herramientas, 150, 172
herramientas de desarrollador, 140
hexadecimal number, 159
hidden element, 239, 266
higher-order function, *see* function, higher-order
hilo, 190, 262
historia, 6
Hières-sur-Amby, 191
hooligan, 314
hora, 152, 154, 156
href attribute, 232, 235
HTML, 227, 308
  structure, 227, 229
html (HTML tag), 228
HTTP, 298, 305, 307, 311
  client, 299, 306
  cliente, 309
  server, 297, 301, 320
http package, 297, 299
HTTPS, 299
https package, 299
human language, 23
hyphen character, 240

id attribute, 232, 240
idempotence, 199, 305
idempotency, 345
identifier, 214
identity, 67
if keyword, 29, 170
  chaining, 30, 36, 329, 330
If-None-Match header, 313, 320, 325
imagen, 233
imagenes, 261
imaginación, 267
IME, 254
img (HTML tag), 233, 238, 261
immutable, 273
implements (reserved word), 26
import keyword, 182
in operator, 65, 110
includes method, 71, 72, 334
indefinido, 134
indentación, 33
index property, 155
index.html, 321
index.js, 292
indexOf method, 75, 76, 98, 120, 151, 165, 334
infinite loop, 48, 146
infinity, 14
infraestructura, 177
inheritance, 104, 117, 119, 146, 304
INI file, 168
ini package, 177, 181, 184, 293
inicialización, 261
inline element, 236, 238
inmutables, 66, 126
inner function, 44
inner loop, 161
innerHeight property, 259
innerWidth property, 259
input, 140, 248, 290, 317
input (HTML tag), 260
insertBefore method, 233
installation, 177
instanceof operator, 118, 146
instancia, 105

integer, 13
integration, 150, 230
interface, 110, 115, 120, 184, 198
  design, 56, 150, 156, 162, 166, 229, 230, 275
  HTTP, 311
  module, 293
  object, 271
interface (reserved word), 26
interfaces, 101, 178
interfaz, 111, 150, 175, 178, 179, 181, 184
internationalization, 170
Internet, 168
interpolation, 16
interpretation, 8, 178, 218, 219, 223
inversion, 152
invoking (of functions), *see* function application
isDirectory method, 304, 344
isEven (exercise), 58, 330
isolation, 101, 175, 178
iterable interface, 113, 334
iterador, 207
iterator interface, 111, 113, 120

Jacques, 60
jardinería, 309
Java, 6
JavaScript, 6
  availability of, 1
  flexibility of, 7
  history of, 6
  puntos débiles, 7
  syntax, 23
  uses of, 8
  versions of, 7
JavaScript console, 8, 17, 28, 139, 145
JavaScript Object Notation, *see* JSON
join method, 99, 108, 293
journal, 61, 64, 66, 72
JOURNAL data set, 71
journalEvents function, 72
JSON, 81, 184, 192, 203, 311, 312, 326, 345
JSON.parse function, 82, 345
JSON.stringify function, 82
JSX, 324
juego, 267–269
juego de plataforma, 267
jugador, 267–269
jump, 5
jump-and-run game, 267
jumping, 268, 285

Kernighan, Brian, 134
key code, 286
key property, 253, 342
keyboard, 253, 268, 285, 286, 289
keydown event, 253, 263, 286, 342
keyup event, 253, 286
kill process, 298
Knuth, Donald, 41

la salida estándar, 291
landscape example, 44
Laozi, 189
lastChild property, 231
lastIndex property, 166, 167
lastIndex property*, 166
lastIndexOf method, 75
latency, 183
lava, 268, 269, 278, 281, 284
Lava class, 273, 284
layout, 236, 239
laziness, 237
Le Guin, Ursula K., 2
leaf node, 229
leak, 289

learning, 2
leerAlmacenamiento function, 193
left (CSS), 242–244, 246
legibilidad, 4
LEGOS, 175
length property
  for array, 63
  for string, 55, 59, 62, 77, 331
lenguaje de programación, 213
lenguaje Egg, 214
lenguajes de programación, 1
less than, 17
let keyword, 24, 25, 43, 67, 79, 81, 135
level, 269, 275, 276, 278, 288
Level class, 269
lexical scoping, 44
leyes De Morgan, 333
library, 230, 293
licencia, 177
limite, 91
line break, 15, 168
line comment, 38, 163
lines of code, 222
link (HTML tag), 280
linked list, 84, 332
linter, 182
Liskov, Barbara, 101
list (exercise), 84, 332
lista de trabajo, 131
listen method, 297, 298
listening (TCP), 297
literal expression, 23, 150, 216, 218
live data structure, 227, 234, 242, 343
live view, 326, 346
lives (exercise), 289
llamada de pila, 208
llaves, 30, 65, 154, 330
load event, 261
local binding, 50, 226, 331
local scope, 43, 223
localhost, 298
localStorage object, 322
locked box (exercise), 149, 336
logging, 139
logical and, 18
logical operators, 18
logical or, 19
long polling, 310, 311, 313, 317, 319, 320, 325
loop, 5, 34, 40, 51, 71, 88, 94, 167
  termination of, 35
loop body, 33, 89
lycanthropy, 60, 68
límite, 159, 165
línea, 24, 169
línea de comandos, 291, 293
líneas, 33

machine code, 3, 224
magia, 213
magic, 103
mailRoute array, 129
maintenance, 177
manejadores, 248
manejo de excepciones, 141
map, 274
map (data structure), 109
Map class, 110, 115, 205
map method, 92, 95, 99, 103, 109, 125, 201, 270
Marcus Aurelius, 248
match method, 155, 167
matching, 151, 157, 158, 166, 173
  algorithm, 158–160
matemáticas, 89
Math object, 58, 62, 78
Math.abs function, 80
Math.acos function, 79

Math.asin function, 79
Math.atan function, 79
Math.ceil function, 80, 282
Math.cos function, 79, 244
Math.floor function, 80, 128, 282
Math.max function, 28, 62, 77, 78
Math.min function, 28, 58, 78
Math.PI constant, 79
Math.random function, 79, 128, 274
Math.round function, 80
Math.sin function, 79, 244, 274, 284
Math.sqrt function, 70, 78, 334
Math.tan function, 79
mathematics, 51
Matrix class, 113
matrix example, 113, 117
MatrixIterator class, 114
max-height (CSS), 278
max-width (CSS), 278
maximum, 28, 78, 94
measuring a robot (exercise), 132, 335
media type, 303
meetup, 309
memoria, 61, 81, 189
memory, 11, 24, 47, 67, 84, 224
  persistence, 327
menú contextual, 252
message event, 262
meta key, 254
metaKey property, 254
method, 63, 74, 102, 298
  HTTP, 298, 306, 311, 314
method call, 102
methods object, 301
mime package, 303
MIME type, 303
minificadores, 184
minimalism, 267
minimum, 28, 58, 78
minimum (exercise), 58, 330
minus, 14, 174
MKCOL method, 307, 345
mkdir function, 307, 345
modification date, 304
modifier key, 254
modular robot (exercise), 187, 337
modularity, 101
module, 187, 275, 314
  design, 184
module loader, 292
module object, 181
modulo operator, 14
modulos CommonJS, 187
monedas, 269
monociclismo, 309
monster (exercise), 289, 344
mouse, 27
mouse cursor, 255
mouse trail (exercise), 266, 342
mousedown event, 251, 255, 257
mousemove event, 256, 257, 263, 264, 266, 342
mouseup event, 255, 257
movimiento, 268
multiplication, 13, 272, 284
multiplier function, 50
mundo, 267
mundo virtual, 122
musical, 267
mutability, 65–67, 126
método, 101, 105, 135
módulo, 175, 292, 293
módulo CommonJS, 187, 338
módulos, 178
módulos CommonJS, 182
módulos ES, 182

namespace, 78
namespace pollution, 78

naming, 26
NaN, 14, 18, 20, 134
navegador, 2, 27, 268, 309
navegadore, 28, 190
navegadores, 9, 183
negation, 17, 19
negritas, 238
neighbors property, 200
nerd, 165
nesting
  in regexps, 161
  of arrays, 70
  of expressions, 23, 215
  of functions, 44
  of loops, 40, 330
  of objects, 228, 231
  of scope, 44
Netscape, 6
network, 197
  speed, 290
network function, 205
new operator, 105
newline character, 15, 40, 152, 163, 168
next method, 113, 207, 334
nextSibling property, 231
nivel, 268, 269
node program, 291
node-fetch package, 299
Node.js, 8, 9, 28, 179, 190, 290–293, 296–299, 301, 303–306, 309–311, 314, 327
node_modules directory, 292, 294
NodeList type, 230, 241
nodeName property, 246
nodeType property, 229, 342, 343
nodeValue property, 231
nodo, 228, 229
nodos de texto, 231
nodos hijos, 229
nodos hoja, 229
nombrado, 4
nombre, 6
not a number, 14
notación, 182
NPM, 177, 179, 181, 183, 185, 187, 292–295, 303, 314, 315, 327, 338
npm program, 293–295, 303
nueces, 73, 74
null, 20, 21, 53, 62, 81, 85, 140
number, 12, 66
  conversion to, 20, 29
  notation, 12, 13
  precision of, 13
  representation, 12
  special values, 14
Number function, 29, 37
number puzzle example, 52
Number.isNaN function, 30
número, 151, 174, 337
número binario, 70
números pare, 31

object, 28, 60, 65, 66, 78, 81, 84, 103, 118, 333
  as map, 274
  creation, 105
  identity, 67
  property, 62
  representation, 81
Object prototype, 104
object-oriented programming, 105, 111, 117, 185
Object.create function, 104, 109, 222
Object.keys function, 65, 85, 205, 333, 341
Object.prototype, 109
objeto, 64, 101, 178
objeto de evento, 250

objeto tipo array, 296
objeto tipo matriz, 258
objetos de evento, 258
obstacle, 281
offsetHeight property, 236, 237
offsetWidth property, 236
on method, 300
onclick attribute, 249
opcional, 153
operador, 214
operador binario, 23
operador unario, 23
operator, 13, 17, 21, 221
  application, 13
optimización, 268
optimization, 52, 57, 237, 263, 278, 297
option (HTML tag), 345
optional argument, 49, 83
ordinal package, 179, 181
organic growth, 175
organization, 175
output, 17, 27, 28, 139, 140, 221
overflow (CSS), 278
overlap, 282
overlay, 240
overriding, 107, 111, 117, 339
overwriting, 306, 308, 318

p (HTML tag), 236
package (reserved word), 26
package manager, 177
package.json, 294, 295
padding (CSS), 277
page reload, 261
pageX property, 255, 258
pageXOffset property, 237
pageY property, 255, 258
pageYOffset property, 237, 259
palabra caracter, 170
palabra clave, 24, 236
palabras clave, 26
Palef, Thomas, 267
paquete, 176, 179, 292, 295
parallelism, 190
parameter, 27, 42, 49, 80, 136
parametros, 181
parent node, 251
parentheses, 14, 158
parentNode property, 231
parse function, 217
parseApply function, 216
parseExpression function, 215
parseINI function, 169, 176
parsing, 82, 134, 169, 213–215, 217, 219, 222, 227, 302, 319
parámetro, 41, 43, 46, 48, 77, 103
parámetro restante, 78
paréntesis, 23, 27, 30, 32, 34, 46, 89, 154, 157, 170, 213, 336
path
  file system, 292, 301
  URL, 301, 302, 311, 314
path package, 303
pathfinding, 129, 185, 203
patrones, 150, 151
patrón, 150, 152, 164
pattern, 151
pausing (exercise), 289, 343
pea soup, 87
percentage, 259
performance, 160, 183, 223, 237, 268, 297
period character, 62, *see* max example, 152, 163, 174
persistence, 327, 345
persistencia, 309
persistent data structure, 124, 126, 132, 138, 344
persistent group (exercise), 132

persistent map (exercise), 335
petición, 310
PGroup class, 132, 335
phase, 273, 274, 284
phi coefficient, 69, 70
phi function, 70, 80
physics, 281, 285
physics engine, 281
pi, 13, 79, 244, 274
PI constant, 79, 244
pila, 64
pila de llamadas, 48
ping request, 200
pipe character, 337
pipe method, 302, 306
pipeline, 184
pixel, 236, 245, 255, 268, 276
pizza, 69, 70
plantilla, 179
plantillas literales, 16
platform game, 289
Plauger, P.J., 134
player, 278, 281, 284, 288
Player class, 272, 284
plus character, 13, 153, 174
Poignant Guide, 23
pointer, 230
pointer event, 251
polling, 248
pollTalks function, 325
polymorphism, 111
pop method, 63, 74
Popper, Karl, 234
porcentaje, 99
port, 298
position, 237
position (CSS), 242, 246, 259, 268, 277, 278
POST method, 312
postMessage method, 262
power example, 42, 49, 51
precedence, 14, 19
precedencia, 14, 240
predicate function, 91, 96, 100
Prefer header, 313, 320, 325
pregunta de entrevista, 40
premature optimization, 52
preventDefault method, 252, 258, 260, 261, 286
previousSibling property, 231
primitiveMultiply (exercise), 148, 336
private (reserved word), 26
private properties, 101
private property, 149
problema de búsqueda, 130
procesador, 189
process object, 291, 303
profiling, 52
program, 23, 29
  nature of, 2
program size, 86, 173, 275
programación, 1
programación asincrónica, 192
programación orientada a objetos, 101, 124
programming
  difficulty of, 2
  history of, 4
  joy of, 1, 3
programming language, 3, 224, 229, 290
  power of, 6
programming style, 3, 24, 33, 37, 275
project chapter, 122, 213, 267
promesa, 212, 340
Promise class, 195, 197–199, 205, 208, 209, 212, 297, 299, 301, 326, 339
Promise.all function, 200, 210, 212, 339

Promise.reject function, 197
Promise.resolve function, 195, 200
promises package, 297
promptDirection function, 145, 146
promptInteger function, 140
propagation, *see* event propagation
property, 62, 64, 102, 103, 111, 115
  assignment, 65
  deletion, 65
  model of, 65
  testing for, 65
propiedad, 28, 62, 68, 105, 107, 112, 134
propiedades, 110
protected (reserved word), 26
prototipo, 104, 106–108
prototipos, 105
prototype, 109, 117, 222, 226, 340, 345
  diagram, 108
prototype property, 106
proyecto para compartir habilidades, 309
pseudorandom number, 79
public (reserved word), 26
public properties, 101
public space (exercise), 308, 345
publishing, 295
punch card, 4
punto de interrupción, 139
punto y coma, 34, 239
pure function, 56, 57, 83, 92, 185
push method, 63, 72, 74, 334
PUT method, 301, 305, 312, 317, 345
Pythagoras, 334
página web, 183
páginas de inicio llamativas, 266

query string, 313, 319
querySelector method, 242, 341
querySelectorAll method, 241
question mark, 19, 153
queue, 208
quotation mark, 15
quoting
  in JSON, 81
  of object properties, 64
quoting style (exercise), 174, 336

rabbit example, 102, 104, 106, 107
radian, 244
radix, 11
raising (exception), 141
random number, 79, 274
random-item package, 337
randomPick function, 127
randomRobot function, 127
range function, 83, 331
rango, 91, 152, 153
rangos, 154
raíz, 228
read-eval-print loop, 291
readability, 6, 38, 51, 56, 141, 175, 219, 280
readable stream, 299, 300, 302, 317
readdir function, 297, 304, 344
readdirSync function, 344
readFile function, 180, 296, 345
readFileSync function, 297, 344
reading code, 8, 122
readStream function, 317, 318
real-time, 248
reasoning, 18
recipe analogy, 87
record, 64
rectangle, 268, 281
recursion, 48, 51, 52, 58, 84, 199, 205, 215, 217, 219, 231, 246, 330, 332, 339, 341

recursión, 336
red, 183, 189
reduce method, 93, 95, 99, 100, 333
ReferenceError type, 226
RegExp class, 150, 164, 344
regexp golf (exercise), 173
regular expression, 151, 152, 161, 163, 166, 307, 314, 315, 341, 344
  alternatives, 158
  backtracking, 159
  boundary, 157
  creation, 150, 164
  escaping, 150, 165, 337
  flags, 154, 161, 165, 337
  global, 161, 166, 167
  grouping, 154, 162
  internationalization, 170
  matching, 158, 166
  methods, 151, 155, 165
  repetition, 153
rejecting (a promise), 196, 199, 209
relación simbiótica, 191
relative path, 181, 292, 301, 345
relative positioning, 242, 243
remainder operator, 14, 35, 329, 330, 342
remote access, 301
removeChild method, 233
removeEventListener method, 249, 343
rename function, 297
renderTalk function, 323
renderTalkForm function, 324
renderUserField function, 323
repeat method, 77, 259
repeating key, 253
repetición, 160, 164
repetition, 54, 153, 263
replace method, 161, 162, 174, 336
replaceChild method, 233, 342
reportError function, 323
request, 297–299, 306
request function, 198, 299, 300
requestAnimationFrame function, 243, 261, 263, 286, 342
requestType function, 199
require function, 179, 180, 187, 292–294, 303, 314
reserved word, 26
resolution, 181, 292
resolve function, 303
resolving (a promise), 195, 196, 199, 209
resource, 301, 316
response, 298, 302, 305
responsiveness, 248, 290
respuesta, 193, 198
retry, 198
return keyword, 42, 47, 105, 206, 330, 333
return value, 28, 42, 140, 333
reuse, 57, 118, 175–177, 293
reverse method, 83
reversing (exercise), 83, 331
rgb (CSS), 277
right-aligning, 246
rmdir function, 304, 307
roadGraph object, 123
roads array, 122
roads module (exercise), 187, 338
robot, 122, 124, 127, 129, 132, 187
robot efficiency (exercise), 132, 335
robustez, 311
rounding, 80, 139, 282
router, 310, 314
Router class, 314, 315
routeRequest function, 204
routeRobot function, 129
routing, 202
rule (CSS), 240, 241
run function, 222

run-time error, 137, 138, 140, 147, 341
runAnimation function, 286, 289
runGame function, 288, 289
runLevel function, 287, 289
running code, 8
runRobot function, 127, 335

sandbox, 8
scaling, 276
scalpel (exercise), 211, 339
scheduling, 208
scientific notation, 13, 174
scope, 44, 50, 218, 221, 225, 226, 340, 341
script, 254
script (HTML tag), 261
SCRIPTS data set, 91, 93, 96, 98, 100
scroll event, 259, 263
scrolling, 252, 259, 278, 279, 286
search method, 165
search problem, 232, 307
search tool (exercise), 307, 344
searching, 158, 160, 165
sección, 168
secuencia, 153
security, 303, 313
seguimiento de la pila, 142
selección, 241
select (HTML tag), 345
semicolon, 23, 24
send method, 193, 198
sendGossip function, 201
sep binding, 303
Separador de vocales Mongol, 170
serialization, 81, 82
server, 297, 298, 300, 301, 314
servidor, 290, 309
set, 152
set (data structure), 120, 132
Set class, 120, 132, 335
set method, 110
setAttribute method, 235, 236
setInterval function, 263
setter, 116
setTimeout function, 192, 208, 262, 263, 320, 326
shared property, 105, 107, 108
shift key, 254
shift method, 74
shiftKey property, 254
short-circuit evaluation, 53, 219, 333
SICP, 213
side effect, 24, 35, 57, 66, 83, 92, 166, 184, 210, 230, 233, 237
sign, 13, 174
sign bit, 13
signal, 11
signo, 337
signo de interrogación, 164
simplicity, 224
simulation, 124, 127, 267, 272, 342
sine, 79, 244, 274, 284
single-quote character, 15, 174
singleton, 133
sintaxis, 213, 214
sintáctico, 228
sistema de archivos, 296
sistema de escritura, 90
sistema de módulos, 178
sitio web, 309
sitios web, 290
skill-sharing project, 311, 314, 321
SkillShareApp class, 326
skipSpace function, 216, 226
slash character, 14, 38, 150, 163, 303, 344
slice method, 75, 76, 91, 234, 331, 340

sloppy programming, 264
smooth animation, 244
socket, 311
solicitud, 193, 198
some method, 96, 100, 201, 315
sondeo, 248
sondeo largo, 310
sorting, 228
source property, 166
special form, 213, 219
special return value, 140, 141
specialForms object, 219
speed, 1
split method, 124, 270
spread operator, 277
square brackets, 62, 78, 152, 331
square example, 41, 45, 46
square root, 70, 78, 334
stack, *see* call stack
stack overflow, 48, 51, 59, 330
standard, 290
standard environment, 27
standard output, 301
startsWith method, 302
stat function, 297, 303, 304, 344
state, 32, 34, 35
  in
    objects, 270
  of application, 278, 328
statement, 23
static (reserved word), 26
static file, 312, 315
static method, 116, 120, 270, 335
Stats type, 304
statSync function, 344
status code, 291
status property, 322
stdout property, 301
stoicism, 248
stopPropagation method, 251
storage function, 195
stream, 298–300, 302, 306, 317
strict mode, 135
string, 15, 61, 63, 66
  indexing, 59, 75, 77, 97, 155
  length, 39, 97
  methods, 76, 155
  notation, 15
  properties, 76
  representation, 16
  searching, 76
String function, 29, 111
strings, 96
strong (HTML tag), 236, 238
structure, 176
structure sharing, 84
style, 238
style (HTML tag), 240
style attribute, 238–240, 275
style sheet, *see* CSS
subclase, 117
submit event, 324, 345
substitution, 57
subtraction, 14, 120
suites de prueba, 138
sum function, 83
summing (exercise), 83, 331
summing example, 5, 86, 93, 222
superclase, 117
suposición, 145
Sussman, Gerald, 213
switch keyword, 36
Symbol function, 111
Symbol.iterator symbol, 113
SymmetricMatrix class, 117
synchronization, 326, 346
synchronous programming, 189, 206, 297, 307
syncState method, 346
syntax, 12, 13, 15, 23, 24, 26, 29, 32,

34, 36, 41, 45, 64, 134, 135, 141, 145, 174
syntax tree, 215, 217, 218, 228
SyntaxError type, 216
símbolo, 112

tabbed interface (exercise), 266, 343
tabindex attribute, 254
tabla, 70, 71
tabla de frecuencias, 69
table, 277
table (HTML tag), 245, 268, 277
table example, 341
tableFor function, 71
talk, 316–318
talkResponse method, 319
talksAbout function, 231
talkURL function, 322
Tamil, 90
tangent, 79
target property, 251
task management example, 74
TCP, 311
td (HTML tag), 246, 277
teclado, 27, 248, 252, 253
teclado virtual, 254
teléfono, 254
temperature example, 116
template, 328, 346
tentacle (analogy), 25, 65, 67
teoría, 139
terminal, 291
termitas, 191
ternary operator, 19, 22, 219
test method, 151
test suite, 137
testing, 132, 137
text, 15
text node, 229, 231, 234, 343
text-align (CSS), 246
TEXT_NODE code, 229, 343
textarea (HTML tag), 263, 345
textContent property, 342
texto, 227, 229, 296
textScripts function, 98, 333
th (HTML tag), 246
then method, 195–197, 200, 339
this, 63, 102, 103, 105, 135
thread, 208
throw keyword, 141, 142, 146, 148, 336
tiempo, 156, 192, 244, 264
Tiempo Unix, 156
time, 281, 284, 287
timeline, 190, 208, 243, 248, 261
timeout, 198, 262, 313, 320
Timeout class, 198
times method, 272
tipo, 12
tipo de solicitud, 194
tipo variable, 137
title, 321
toLowerCase method, 63, 246
tool, 184, 295
top (CSS), 242–244, 246
top-level scope, *see* global scope
toString method, 103, 104, 108, 109, 111, 300
touch, 257
touchend event, 257
touches method, 281
touches property, 258
touchmove event, 257
touchstart event, 257
toUpperCase method, 63, 137, 246, 300
tr (HTML tag), 246, 277
trackKeys function, 286, 289
transpilation, 224
traversal, 159

trial and error, 139, 285
triangle (exercise), 39, 329
trigonometry, 244
trigonometría, 79
trim method, 76, 270
truco, 182
true, 17
try keyword, 142, 144, 200, 336
type, 17, 118
type checking, 137, 183
type coercion, 20, 21, 29
type property, 214, 250
typeof operator, 17, 85, 333
TypeScript, 137

unary operator, 17
uncaught exception, 145, 197
undefined, 20, 21, 25, 42, 48, 62, 65, 81, 135, 140
underline, 238
underscore character, 26, 37, 101, 157, 164
Unicode, 16, 18, 90, 96, 152, 170, 171
  property, 171
unidades de código, 96
uniformidad, 214
uniqueness, 240
unit (CSS), 245, 259
Unix, 304, 306, 307
unlink function, 297, 304
unshift method, 74
upcasing server example, 300
updated method, 317, 318, 320, 345
upgrading, 177
URL, 299, 312, 322
url package, 302, 319
urlToPath function, 302
usability, 253
use strict, *see* strict mode
user experience, 248, 323
user interface, 145
UTF16, 16, 96
UTF8, 296, 297

validation, 140, 147, 213, 280, 317, 318
valor de retorno, 194
valores, 11, 194
var keyword, 26, 42, 43, 81
variable, 4, *see* binding
variable contador, 342
Vec class, 120, 270, 271, 284
vector (exercise), 120, 334
velocidad, 3
vencimiento, 311
ventana, 251, 257
verbosity, 47, 190
versionado semántico, 295
versión, 177, 294, 295
viewport, 278, 280
VillageState class, 124
VillaPradera, 122
vinculaciones, 45, 65, 81, 145, 182
vinculación, 32, 34, 40–42, 67, 168
virtual world, 124, 127
vista en vivo, 311
vocabulario, 41, 86
vocabulary, 41, 87
void operator, 26
volatile data storage, 11

waitForChanges method, 320
waiting, 192
wave, 274, 284
Web, *see* World Wide Web
web application, 6
web browser, *see* browser
web worker, 262
WebDAV, 307

website, 308
WebSockets, 311
weekDay module, 178
weekend project, 308
weresquirrel example, 60, 64, 66, 68, 72, 74
while loop, 5, 32, 34, 55, 167
whitespace, 33, 37, 76, 152, 170, 226, 340
  in HTML, 232, 343
  in URLs, 312
  trimming, 270
why, 23
window, 260
window object, 249
with statement, 136
wizard (mighty), 4
word boundary, 157
word character, 157
World Wide Web, 6, 81
writable stream, 298, 299, 301, 302
write method, 298, 299
writeFile function, 296, 300, 345
writeHead method, 298
writing code, 8, 122
WWW, *see* World Wide Web

XML, 230

yield (reserved word), 26
yield keyword, 207
your own loop (example), 100
Yuan-Ma, 11, 175, 290

Zawinski, Jamie, 150
zero-based counting, 59, 61, 156
zeroPad function, 56
zona horaria, 156

ángulo, 244
árbol, 104, 214, 217, 228, 229
árbol de sintaxis, 214
índice, 61